Sylvan: Multi-core Decision Diagrams

Tom van Dijk* and Jaco van de Pol

Formal Methods and Tools, University of Twente, The Netherlands
{dijkt,vdpol}@cs.utwente.nl

Abstract Decision diagrams such as binary decision diagrams and multi-valued decision diagrams play an important role in various fields, including symbolic model checking. An ongoing challenge is to develop data-structures and algorithms for modern multi-core architectures. The BDD package Sylvan provides one contribution by implementing parallelized BDD operations and thus allowing sequential algorithms to exploit the power of multi-core machines.
We present several extensions to Sylvan. We implement parallel operations on list decision diagrams, a variant of multi-valued decision diagrams that is useful for symbolic model checking. We also substitute several core components of Sylvan by new designs, such as the work-stealing framework, the unique table and the operation cache. Furthermore, we combine parallel operations with parallelization on a higher level, by partitioning the transition relation. We show that this results in an improved speedup using the model checking toolset LTSmin. We also demonstrate that the parallelization of symbolic model checking for explicit-state modeling languages with an on-the-fly next-state function, as supported by LTSmin, scales well.

1 Introduction

A core problem in model checking is that space and time requirements increase exponentially with the size of the models. One method to alleviate this problem is symbolic model checking, where sets of states are stored in binary decision diagrams (BDDs). Another method uses parallel computation, e.g., in computer systems with multiple processors. In [9,11], we combined both approaches by parallelizing BDD operations in the parallel BDD library Sylvan.

In the literature, there is some early work involving parallel BDD operations [18,27,23]. Alternative approaches for parallel symbolic reachability use partitioning strategies [26,15]. Also saturation, an optimal iteration strategy, was parallelized using Cilk [7,13]. More recently, a thesis on JINC [24] describes a multi-threaded extension, but does not actually parallelize the BDD operations. Also, a recent BDD implementation in Java called BeeDeeDee [21] allows execution of BDD operations from multiple threads, but does not parallelize the BDD operations. See also [11] for an overview of earlier approaches to parallelizing symbolic model checking and/or binary decision diagrams.

* The first author is supported by the NWO project MaDriD, grant nr. 612.001.101

In the current paper, we present several extensions to Sylvan, in particular integration with the work-stealing framework Lace, an improved unique table, and the implementation of operations on list decision diagrams (LDDs), a variant of multi-valued decision diagrams (MDDs) useful in symbolic model checking.

We also investigate applying parallelism on a higher level than the BDD operations. Since calculating the full transition relation is expensive in symbolic model checking, our model checking toolset LTSmin [3,20,10,17] has the notion of transition groups, which disjunctively partition the transition relations. We exploit the fact that partitioned transition relations can be applied in parallel and show that this strategy results in improved scalability.

In addition, LTSmin supports learning transition relations on-the-fly, which enables the symbolic model checking of explicit-state models, such as Promela, DVE and mCRL2 models. We implement a specialized operation `collect`, which is a combination of `enumerate` and `union`, to perform parallel transition learning and we show that this results in good parallel performance.

This paper is organized as follows. We review background knowledge about BDDs, MDDs and LDDs in Section 2, as well as background information on symbolic model checking and LTSmin. Section 3 discusses the design of our parallel library Sylvan, with an emphasis on the new unique table and the implementation of LDD operations. Section 4 introduces parallelism on the algorithmic level in the model checking toolset LTSmin in order to run parallel symbolic on-the-fly reachability. Section 5 shows the results of several experiments using the BEEM database of explicit-state models to measure the effectiveness of our approach. Finally, Section 6 summarizes our findings and reflections.

2 Preliminaries

2.1 Symbolic Reachability

In model checking, we create abstractions of complex systems to verify that they function according to certain properties. Systems are modeled as a set of possible states of the system and a set of transitions between these states. A core component of model checking is state-space generation using a reachability algorithm, to calculate all states reachable from the initial state of the system.

One major problem in model checking is the size of the transition system. The memory required to store all explored states and transitions increases exponentially with the size of the models. One way to deal with this is symbolic model checking [6], which represents states as sets rather than storing them individually.

An efficient method to store sets of states uses Boolean functions $S\colon \mathbb{B}^N \to \mathbb{B}$. Every state for which the function S is `true` is in the set. Boolean functions can be stored efficiently using binary decision diagrams (BDDs). Similarly, states can also be represented using functions $S\colon \mathbb{N}^N \to \mathbb{B}$, which can be stored using multi-valued decision diagrams (MDDs) or list decision diagrams (LDDs).

2.2 Binary decision diagrams and multi-valued decision diagrams

Binary decision diagrams (BDDs) were introduced by Akers [1] and developed by Bryant [5].

Definition 1 (Binary decision diagram). *An (ordered) BDD is a directed acyclic graph with the following properties:*

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p has a variable label x_i and two outgoing edges, labeled 0 and 1; we write $lvl(p) = i$ and $p[v] = q$, where $v \in \{0, 1\}$.*
3. *For each edge from node p to non-terminal node q, $lvl(p) < lvl(q)$.*
4. *There are no duplicate nodes, i.e.,*
   $\forall p \forall q \cdot (lvl(p) = lvl(q) \wedge p[0] = q[0] \wedge p[1] = q[1]) \rightarrow p = q$.

Furthermore, either of two reductions ensures canonicity:

Definition 2 (Fully-reduced/Quasi-reduced BDD). *Fully-reduced BDDs forbid redundant nodes, i.e., nodes with $p[0] = p[1]$. Quasi-reduced BDDs keep all redundant nodes, i.e., skipping levels is forbidden.*

Multi-valued decision diagrams (MDDs, also called multi-way decision diagrams) are a generalization of BDDs to the integer domain [16].

Definition 3 (Multi-valued decision diagram). *An (ordered) MDD is a directed acyclic graph with the following properties:*

1. *There is a single root node and terminal nodes 0 and 1.*
2. *Each non-terminal node p has a variable label x_i and n_i outgoing edges, labeled from 0 to $n_i - 1$; we write $lvl(p) = i$ and $p[v] = q$, where $0 \leq v < n_i$.*
3. *For each edge from node p to non-terminal node q, $lvl(p) < lvl(q)$.*
4. *There are no duplicate nodes, i.e.,*
   $\forall p \forall q \cdot (lvl(p) = lvl(q) \wedge \forall v \cdot p[v] = q[v]) \rightarrow p = q$.

Similar to BDDs, fully-reduced and quasi-reduced MDDs can be defined:

Definition 4 (Fully-reduced/Quasi-reduced MDD). *Fully-reduced MDDs forbid redundant nodes, i.e., nodes where for all v, w, $p[v] = p[w]$. Quasi-reduced MDDs keep all redundant nodes, i.e., skipping levels is forbidden.*

See Fig. 1 for an example of an MDD representing a set.

In [8], Ciardo et al. mention advantages of quasi-reduced MDDs: edges that skip levels are more difficult to manage and quasi-reduced MDDs are cheaper than alternatives to keep saturation operations correct. In [2], Blom et al. prefer quasi-reduced MDDs since the set of possible values at each level is dynamic and extending the set of values requires an update of every diagram in a fully-reduced setting, while having no impact in the quasi-reduced setting.

A typical method to store MDDs in memory is to store the variable label x_i plus an array holding all n_i edges, e.g., in C: `struct node { int lvl; struct node* edges[0]; }` as in [22]. New nodes are dynamically allocated using `malloc` and a hash table ensures that no duplicate MDD nodes are created.

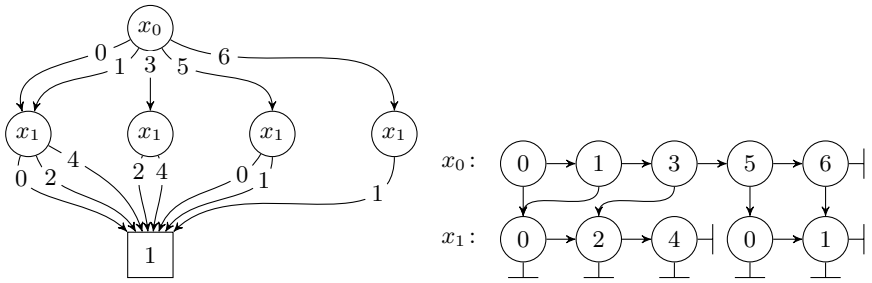

Figure 1. Edge-labeled MDD hiding paths to 0 (left) and LDD (right) representing the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$. For simplicity, we hide paths to 0 and 1; the $p[x_i > v]$ edge of the last node in each "linked list" goes to 0, and every $p[x_i = v]$ edge on the last level goes to 1.

Alternatively, one could use a large `int[]` array to store all MDDs (each MDD is represented by $n_i + 1$ consecutive integers) and represent edges to an MDD as the index of the first integer. In [8], the edges are stored in a separate `int[]` array to allow the number of edges n_i to vary.

2.3 List decision diagrams

Implementations of MDDs that use arrays to implement MDD nodes have two disadvantages. (1) For *sparse* sets (where only a fraction of the possible values are used) using arrays is a waste of memory. (2) MDD nodes typically have a variable size, complicating memory management. An alternative method uses list decision diagrams (LDDs), which can be understood as a linked-list representation of quasi-reduced MDDs. LDDs were initially described in [2, Sect. 5].

Definition 5 (List decision diagram). *A List decision diagram (LDD) is a directed acyclic graph with the following properties:*

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p is labeled with a value v, denoted by $val(p) = v$, and has two outgoing edges labeled $=$ and $>$ that point to nodes denoted by $p[x_i = v]$ and $p[x_i > v]$.*
3. *For all non-terminal nodes p, $p[x_i = v] \neq 0$ and $p[x_i > v] \neq 1$.*
4. *For all non-terminal nodes p, $val(p[x_i > v]) > v$.*
5. *There are no duplicate nodes.*

An LDD can be constructed from a quasi-reduced MDD by dropping all edges to 0 and creating an LDD node for each edge, using the edge label as the value in the LDD node. In a quasi-reduced BDD/MDD, every path from the root to a terminal encounters every variable (in the same order). Hence the variable label x_i follows implicitly from the depth of the node. We therefore do not store it in the LDD nodes either. The root node is at level 0, non-terminal nodes following

```
1 def reachable(initial, trans, K):
2     states = {initial}
3     next = states
4     while next != ∅:
5         for k in (0..K-1):
6             learn(next, k)
7             successors[k] = relprod(next, trans[k])
8             successors[k] = minus(successors[k], states)
9         next = union(successors[0], ..., successors[K-1])
10        states = union(states, next)
11    return states
```

Figure 2. Symbolic on-the-fly reachability algorithm with transition groups: compute the set of all states reachable from the initial state. The transition relations are updated on-the-fly (line 6) and the algorithm relies on BDD operations `relprod` (relational product), `minus` ("diff") and `union` ("or"). See Fig. 9 for `learn`.

>-edges have the same level and non-terminal nodes following =-edges have the next level. See Fig. 1 for an example of an MDD and an LDD representing the same set of integer pairs.

2.4 LTSmin and partitioned transition relations

The model checking toolset LTSmin[1] provides a language independent Partitioned Next-State Interface (Pins), which connects various input languages to model checking algorithms [3,20,10,17]. In Pins, states are vectors of N integer values. Furthermore, transitions are distinguished in disjunctive *transition groups*. The transition relation of each transition group is defined on a subset of the entire state vector, enabling efficient encoding of transitions that only affect a few integers of the state. For example, in a model of a software program, there could be a separate transition group for every line of source code.

Every language module implements a NextState function, which computes the successors of a state for each transition group. Algorithms in LTSmin thus learn new transitions on-the-fly. The reachability algorithm for symbolic model checking using BDD operations is given in Fig. 2.

3 Sylvan: Parallel BDD and LDD operations

In [11], we implemented Sylvan[2], a parallel BDD package, which parallelizes BDD operations using lock-less data structures and work-stealing. Work-stealing [4] is a task-based load balancing method that involves breaking down a calculation into an (implicit) tree of (small) tasks. Independent subtasks are stored in queues and idle processors steal tasks from the queues of busy processors. Most BDD operations in Sylvan are implemented as recursive tasks, where operations are

[1] Available from https://github.com/utwente-fmt/ltsmin (open source).
[2] Available from https://github.com/utwente-fmt/sylvan (open source).

```
1 def ite(A,B,C):
2     if A = 1: return B
3     if A = 0: return C
4     result = cache_lookup(A,B,C)
5     if result = None:
6         x = min(var(A), var(B), var(C))
7         do in parallel:
8             Rlow = ite(low(A, x), low(B, x), low(C, x))
9             Rhigh = ite(high(A, x), high(B, x), high(C, x))
10        result = uniqueBDDnode(x, Rlow, Rhigh)
11        cache_store(A,B,C,result)
12    return result
```

Figure 3. The `ite` algorithm calculating $(A \to B) \wedge (\overline{A} \to C)$ is used to implement binary operations like `and`, `or`. The recursive calls to `ite` are executed in parallel. BDDs are automatically fully-reduced by the `uniqueBDDnode` method using a hash table.

```
1 def uniqueBDDnode(var, edgelow, edgehigh):
2     if edgelow = edgehigh: return edgelow
3     node = {var, edgelow, edgehigh}
4     try:
5         return nodestable.insert-or-find(node)
6     catch TableFull:
7         garbagecollect()
8         return nodestable.insert-or-find(node)
```

Figure 4. The `uniqueBDDnode` method creates a BDD node using the hash table `insert` method (Fig. 6) to ensure that there are no duplicate nodes. Line 2 ensures that there are no redundant nodes.

performed on the two subtasks in parallel, and the final result is computed using a hash table. See Fig. 3 for the `ite` algorithm. Other algorithms such as relational product and existential quantification are implemented similarly. To ensure that the results of BDD operations are canonical, reduced BDDs, they use a method `uniqueBDDnode` that employs a hash table as in Fig. 4.

We substituted the work-stealing framework Wool [14], that we used in the original version of Sylvan, by Lace [12], which we developed based on some ideas to minimize interactions between different workers and with memory. Lace is based around a novel work-stealing queue, which is described in detail in [12].

The parallel efficiency of a task-based parallelized algorithm depends in part on the contents of each task. If parallelized tasks mainly perform processor calculations and depend on many subtasks in a predictable or regular fashion, then they result in good speedups. However, if the number of subtasks is small and the subtasks are relatively shallow, i.e., the "task tree" has a low depth, then parallelization is more difficult. BDD tasks typically "spawn" only one or two subtasks for parallel execution, depending on the operation and the input BDDs.

BDD operations are also memory-intensive, since they consist mainly of operations on two data structures: a *unique table* that stores the unique BDD nodes and an *operation cache* that stores the results of BDD operations. The unique table is a hash table with support for garbage collection. The operation cache is a

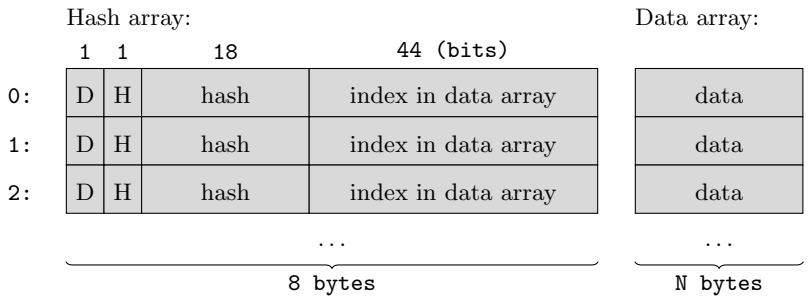

Figure 5. Layout of the new lock-less hash table using a separate hash array `h` and data array `d`. `h`$[n].D$ controls whether `d`$[n]$ is used; `h`$[n].H$ controls whether `h`$[n]$ is used, i.e., the hash and index values correspond with an existing entry in the hash table. Every modification of `h`$[n]$ must be performed using a `compare_and_swap` operation.

simplified hash table that overwrites on collision. Hence the design of concurrent scalable hash tables is crucial for a scalable BDD implementation.

3.1 Lock-less hash table

In parallel programs, memory accesses are typically protected against race conditions using locking techniques. Locking severely cripples parallel performance, therefore Sylvan implements lock-less data structures that rely on the atomic `compare_and_swap` (`cas`) memory operation and short-lived local `cas`-locks to ensure parallel correctness as well as scalable performance.

Compared to [11], we implemented a new hash table based on the lock-less hash table presented in [19]. The new hash table consists of a *hash array* and a *data array*, as in Fig. 5. The data array simply stores fixed-sized data, such as BDD nodes and LDD nodes. We preallocate this data array to avoid losing scalability due to complex memory allocation management systems. Data can be stored at any position in the data array, and this position is recorded in the index field of the hash array. The advantage of storing data at any position is that this allows rehashing all BDD nodes without changing their position in the data array, which is important for efficient garbage collection.

The state of the buckets is manipulated using straight-forward `cas`-operations. Inserting data consists of three steps: searching whether the data is already in the table, claiming a bucket in the data array to store the data, and inserting the hash in the hash array. See Fig. 6.

First, the algorithm obtains a hash value of the data and a *probe sequence* similar to [19], which makes optimal use of memory management in modern systems. See further [19] for more details. The algorithm checks whether one of the hash buckets in the probe sequence already contains an entry with the same hash and with matching data in the data array. If so, it terminates.

If an empty hash bucket is reached, it searches for an empty data bucket (where D=0 in the hash array) and uses `cas` to set D=1, then writes the data.

```
1 def insert-or-find(data):
2    h = calculate_hash(data)
3    ps = probe_sequence(data):
4    while ps != empty:
5       s, ps = head(ps), tail(ps)
6       V = hasharr[s]
7       if V.H = 0: goto EmptyBucketFound
8       if V.hash = h && dataarr[V.index] = data: return V.index
9    raise TableFull # abort: table full!

11 label EmptyBucketFound:
12   for d in (0..N): # note: traverse in smart order
13      W = hasharr[d]
14      if W.D=0:
15         if cas(hasharr[d],W,W[D=1]):
16            dataarr[idx]=data
17            goto EmptyDataSlotFound

19 label EmptyDataSlotFound:
20   while ps != empty: # note: continue same probe sequence
21      V = hasharr[s]
22      if V.H=0:
23         if cas(hasharr[s],V,V[H=1,hash=h,index=d]): return d
24      else:
25         if V.H = h && dataarr[V.index] = data:
26            W = hasharr[d]
27            while !cas(hasharr[d],W,W[D=0]): W = hasharr[d]
28            return V.index
29         s, ps = head(ps), tail(ps)
30   W = hasharr[d]
31   while !cas(hasharr[d],W,W[D=0]): W = hasharr[d]
32   raise TableFull # abort: table full!
```

Figure 6. Algorithm for parallel `insert` of the lock-less hash table.

The position in the data array can be any position. In practice, we record the empty data bucket of the previous `insert` call in a thread-specific variable, and continue from there. Initial values for this thread-specific variable are chosen such that all threads start at a different position in the data array.

After adding the entry in the data array, the algorithm continues with the probe sequence, starting where it found the empty hash bucket in the first step, to search either a matching hash with matching data (written by a concurrent thread), or an empty hash bucket. In the first case, the algorithm releases the data bucket by setting D=0 using `cas`. In the second case, it uses `cas` to set the values H, hash and index at once.

3.2 Sylvan API

Sylvan is released under the Apache 2.0 License, which means that anyone can freely use it and extend it. It comes with an example of a simple BDD-based reachability algorithm, which demonstrates how to use Sylvan to "automatically" parallelize sequential algorithms.

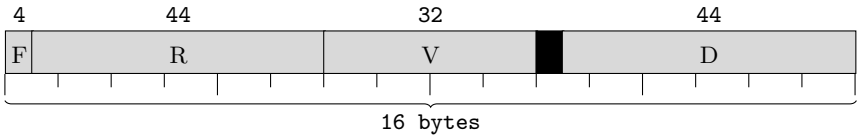

Figure 7. Layout of an LDD node in memory. The F field is reserved for flags, such as "marked", and to indicate possible special node types. The R field contains the index of the LDD node $p[x_i > v]$ and the D field contains the index of the LDD node $p[x_i = v]$ in the LDD node table. The V field contains the 32-bit v value.

To use Sylvan, simply include `sylvan.h` or `lddmc.h` and initialize Lace and Sylvan. There is an API that exposes familiar BDD algorithms, such as `ite`, `exists`, `constrain`, `compose`, `satcount` and `relprod`, a specialized relational product for paired (x_i, x'_i) variables. There is also functionality for drawing DOT graphs and writing/loading BDDs to/from disk. It is easy to add new algorithms by studying the implementation of current algorithms.

Sylvan is distributed with the work-stealing framework Lace. Lace can be integrated in other projects for out-of-the-box task-based parallelism. There are additional methods to integrate Lace worker threads with existing parallel approaches. Furthermore, we developed bindings for Java JNI[3] and Adam Walker developed bindings for Haskell[4], allowing parallelization of algorithms developed in those languages. Extending Sylvan with other types of decision diagrams requires copying files `sylvan.h` and `sylvan.c` and modifying the new files for the different algorithms, similar to what we did with LDDs.

3.3 LDDs in Sylvan

We extended Sylvan with an implementation of LDDs and various LDD algorithms. To represent LDD nodes in memory we use the layout described in Fig. 7. The size of each LDD node is 16 bytes and we allocate 32 bits to hold value v, i.e., the integer values of the state vector in Pins. In our design, 44 bits are reserved to store edges, which is sufficient for up to 2^{44} LDD nodes, i.e., 256 terabytes of just LDD nodes.

We implemented various LDD operations that are required for model checking in LTSmin, especially `union`, `intersection`, `minus`, `project` (existential quantification), `enumerate` and `relprod` (relational product). These operations are all recursive and hence trivial to parallelize using the work-stealing framework Lace and the datastructures earlier developed for the BDD operations.

4 Parallelism in LTSmin

4.1 Parallel symbolic reachability

Even with parallel operations, parallel scalability of model checking in LTSmin is limited, especially in smaller models, when the size of "work units" (between

[3] Available from https://github.com/utwente-fmt/jsylvan.
[4] Available from https://github.com/adamwalker/sylvan-haskell.

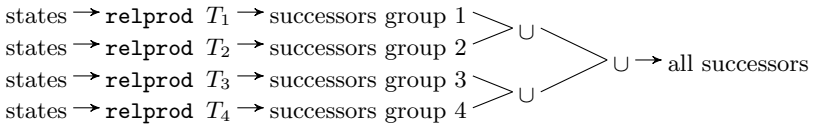

Figure 8. Schematic overview of parallel symbolic reachability. Note that the `relprod` and ∪ operations are also parallel operations internally.

```
1 def learn(states, i):
2     shorts = project(states, vars[i])
3     shorts = minus(shorts, visited[i])
4     visited[i] = union(visited[i], shorts)
5     enumerate(shorts, NextStateWrapper[i])
```

Figure 9. On-the-fly learning in LTSmin; `enumerate` calls NextStateWrapper[i] for each "short" state, which adds new transitions to `trans[i]`.

```
1 def learn_par(states, i):
2     shorts = project(states, vars[i])
3     shorts = minus(shorts, visited[i])
4     visited[i] = union(visited[i], shorts)
5     temp = collect(shorts, NextStateWrapperPar[i])
6     trans[i] = union(trans[i], temp)
```

Figure 10. Parallel on-the-fly learning in LTSmin. The `collect` method combines `enumerate` and `union`.

sequential points) is small and when the amount of parallelism in the work units is insufficient. Experiments in [11] demonstrate this limitation.

This is expected: if a parallel program consists of many small operations between sequential points, then we expect limited parallel scalability. If there are relatively few independent tasks in the "task tree" of a computation, then we also expect limited parallel scalability.

Since LTSmin partitions the transition relation in transition groups (see Section 2.4), many small BDD operations are executed in sequence, for each transition group. We propose to calculate these operations in parallel and merge their results pairwise, as in Fig. 8. In Fig. 2, this corresponds to executing lines 7 and 8 in parallel. This strategy decreases the number of sequential points and thus increases the size of "work units". It also increases the amount of parallelism in the "task tree". We therefore expect improved parallel scalability.

4.2 Parallel on-the-fly learning

As described in Section 2.4, algorithms in LTSmin learn new transitions on-the-fly, using a NextState function. The implementation of the `learn` algorithm used in Fig. 2, is given in Fig. 9. In LTSmin, the transition relation of each transition group is only defined on a subset of the variables in the state vector. First the set of states is "projected" (using existential quantification) such that it is defined only on the variables relevant in transition group `i`. The `visited`

```
1 def collect(states, callback, vec={}):
2     if states = 1: return callback(vec)
3     if states = 0: return ∅
4     do in parallel:
5         R0 = collect(follow-0(states), callback, vec+{0})
6         R1 = collect(follow-1(states), callback, vec+{1})
7     return union(R0, R1)
```

Figure 11. The parallel `collect` algorithm (BDD version) combining `enumerate` and `union`. The callback is called for every state and the returned set is pairwise merged .

set is used to remove all "short states" that were seen before. The `enumerate` method enumerates all states described by the BDD or LDD, i.e., all variable assignments that result in 1. For each state, it calls the supplied callback NextStateWrapper[i]. This method performs a `union` with `trans[i]` and every single discovered transition one by one. Note that this is not thread-safe.

Similar to calculating the relational product for every transition group in parallel, we can perform on-the-fly transition learning for every transition group in parallel. However, there are more opportunities for parallelism.

In Fig. 9, the `project` (existential quantification), `minus` ("diff") and `union` ("or") operations are already parallelized. The `enumerate` method is trivial to parallelize, but the callback wrappper is not thread-safe. We substituted this implementation by a new design that uses a method `collect`. See Fig. 10. The NextStateWrapperPar callback in Fig. 10 adds all transitions for a state to a small temporary decision diagram and returns this decision diagram to the caller. The method `collect` (Fig. 11) performs enumeration in parallel (lines 4–6), and performs a `union` on the results of the two subtasks.

This method works in LTSmin for all language modules that are thread-safe, and has been tested for mCRL2 and DVE models.

5 Experimental evaluation

In the current section, we evaluate the presented LDD extension of Sylvan, and the application of parallelization to LTSmin. As in [11], we base our experimental evaluation mostly on the BEEM model database [25], but in contrast to [11], we use the entire BEEM model database rather than a selection of models. We perform these experiments on a 48-core machine, consisting of 4 AMD Opteron™ 6168 processors with 12 cores each and 128 GB of internal memory.

5.1 Fully parallel on-the-fly symbolic model checking

We perform symbolic reachability using the LTSmin toolset using the following command: `dve2lts-sym --order=<order> --vset=lddmc -rgs <model>.dve`.

We also select as size of the unique table 2^{30} buckets and as size of the operation cache also 2^{30} buckets. Using parameter `--order` we either select the `par-prev` variation or the `bfs-prev` variation. The `bfs-prev` variation does

Experiment	T_1	T_{16}	T_{24}	T_{32}	T_{48}	Speedup T_{48}/T_1
`blocks.4` (par)	629.54	41.61	29.26	23.04	16.58	38.0
`blocks.4` (bfs)	630.04	45.88	33.24	27.01	21.69	29.0
`telephony.8` (par)	843.70	58.17	41.17	32.76	24.68	34.2
`telephony.8` (bfs)	843.06	66.28	47.91	39.17	31.10	27.1
`lifts.8` (par)	377.52	25.92	18.68	15.18	12.03	31.4
`lifts.8` (bfs)	377.36	36.61	30.06	27.68	26.11	14.5
`firewire_tree.1` (par)	16.40	1.09	0.97	0.94	0.99	16.5
`firewire_tree.1` (bfs)	16.43	11.24	11.12	11.36	11.35	1.4
Sum of all `par-prev`	20756	1851	1552	1403	1298	16.0
Sum of all `bfs-prev`	20745	3902	3667	3625	3737	5.6

Figure 12. Results of running symbolic reachability on 269 models of the BEEM database. Each value T_n is the result of at least 3 measurements and is in seconds.

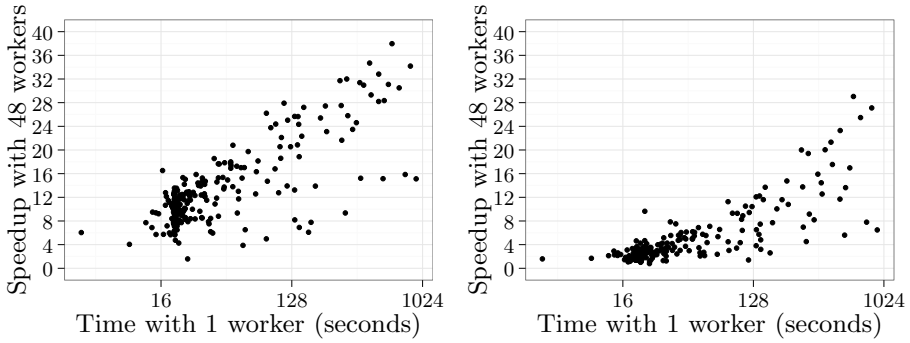

Figure 13. Results of all models in the BEEM database that did not time out, with fully parallel learning and parallel transition groups (`par-prev`) on the left, and only parallel BDD operations (`bfs-prev`) on the right.

not have parallelism in LTSmin, but it uses the parallelized LDD operations, including `collect`. This means that there is parallel learning, but only for one transition group at a time. In the `par-prev` variation, learning and calculating the successors are performed for all transition groups in parallel.

We measure the time spent to execute symbolic reachability, excluding time spent initializing LTSmin. We use a timeout of 1200 seconds. Of all models in the BEEM database, only 7 timed out: `collision.6`, `driving_phils.3`, `driving_phils.5`, `elevator.5`, `frogs.4`, `hanoi.3`, and `public_subscribe.5`. We present here the results for the remaining 269 models. Each benchmark is performed at least 3 times.

See Fig. 12 for the results of this benchmark. Model `blocks.4` results in the highest speedup with `par-prev`, which is 38.0x. We also highlight the model `lifts.8` which has a speedup of 14.5x with `bfs-prev` and more than twice as

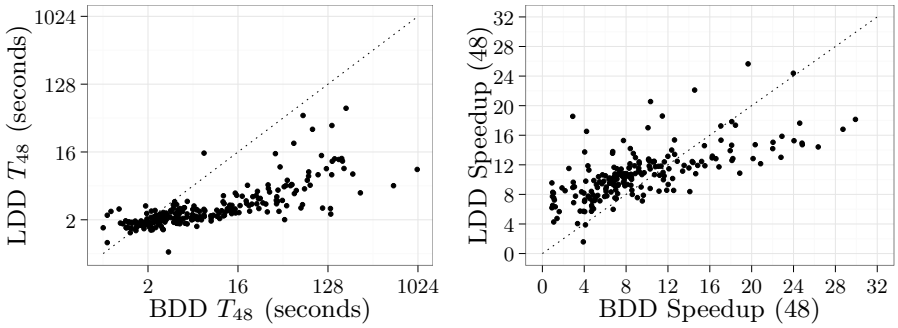

Figure 14. Results of all models that did not time out for both BDDs and LDDs, comparing time with 48 workers (left) and obtained speedups (right).

high with `par-prev`. Fig. 13 shows that "larger" models are associated with a better parallel performance.

Fig. 13 also shows that adding parallellism on the algorithmic level benefits the parallel performance of many models. One of the largest improvements was obtained with the `firewire_tree.1` model, which went from 1.4x to 16.5x. We conclude that the bottleneck was the lack of parallelism.

In addition, Fig. 12 shows that the overhead between the "sequential" `bfs-prev` and "parallel" `par-prev` versions is negligible. Taking the time spent for the entire benchmark set, we see that the speedup of the entire benchmark is 16.0x for the fully parallelized version. For all models, the speedup improves with `par-prev`. The only exception is `peg_solitaire.1`, for which $T_{48} = 2.38$ with `par-prev`, and $T_{48} = 2.35$ with `bfs-prev`, which is within measurement error.

5.2 BDDs and LDDs

We now compare the performance of our multi-core BDD and LDD variants. We do this for the `par-prev` algorithm. Fig. 14 shows that the majority of models, especially larger models, are performed up to several orders of magnitude faster using LDDs. The most extreme example is model `frogs.3`, which has for BDDs $T_1 = 989.40$, $T_{48} = 1005.96$ and for LDDs $T_1 = 61.01$, $T_{48} = 9.36$. Some models are missing that timed out for BDDs but did not time out for LDDs, for example model `blocks.4`.

6 Conclusions

In the current paper we presented several modifications to Sylvan, such as a new hash table implementation for Sylvan and the replacement of the work-stealing framework by Lace. We also discussed how we extended Sylvan to implement parallel LDD operations and the specialized BDD/LDD method `collect` that parallelizes on-the-fly transition learning. We measured the performance of this

implementation using symbolic reachability in LTSmin applied to models from the BEEM model benchmark database. This resulted in a speedup of up to 38.0x when also applying parallelism on the algorithmic level in LTSmin, or up to 29.0x when just using parallelized operations in an otherwise sequential symbolic reachability algorithm.

BDDs and other decision diagrams are also important in other domains. We conclude that sequential algorithms benefit from "automatic" parallelization using parallel BDD and LDD operations in Sylvan. We also conclude that additional parallelism at the algorithmic level results in significant improvements.

Our parallel BDD package is open-source and publicly available online and is easy to integrate with existing software, also using Java JNI bindings and Haskell bindings.

References

1. Akers, S.: Binary Decision Diagrams. IEEE Trans. Computers C-27(6), 509–516 (june 1978)
2. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium. LNCS, vol. 5160, pp. 81–95. Springer (2008)
3. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Proc. of the 22nd int. conf. on Computer Aided Verification. pp. 354–359. CAV'10, Springer-Verlag (2010)
4. Blumofe, R.D.: Scheduling multithreaded computations by work stealing. In: FOCS. pp. 356–368. IEEE Computer Society (1994)
5. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers C-35(8), 677–691 (aug 1986)
6. Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (apr 1994)
7. Chung, M.Y., Ciardo, G.: Saturation NOW. In: QEST. pp. 272–281. IEEE Computer Society (2004)
8. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: Saturation Unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer (2003)
9. van Dijk, T.: The Parallelization of Binary Decision Diagram operations for model checking. Master's thesis, University of Twente, Dept. of C.S. (April 2012)
10. van Dijk, T., Laarman, A.W., van de Pol, J.C.: Multi-core and/or Symbolic Model Checking. ECEASST 53 (2012)
11. van Dijk, T., Laarman, A.W., van de Pol, J.C.: Multi-Core BDD Operations for Symbolic Reachability. In: 11th International Workshop on Parallel and Distributed Methods in verifiCation. ENTCS, Elsevier (2012)
12. van Dijk, T., van de Pol, J.C.: Lace: non-blocking split deque for work-stealing. In: 7th International Euro-Par Workshop on Multi-/Many-core Computing Systems. pp. 206–217. Springer International Publishing (2014)
13. Ezekiel, J., Lüttgen, G., Ciardo, G.: Parallelising symbolic state-space generators. In: CAV. LNCS, vol. 4590, pp. 268–280 (2007)
14. Faxén, K.F.: Efficient work stealing for fine grained parallelism. In: 2010 39th International Conference on Parallel Processing (ICPP). pp. 313–322. IEEE Computer Society, Los Alamitos, CA, USA (sept 2010)

15. Grumberg, O., Heyman, T., Schuster, A.: A work-efficient distributed algorithm for reachability analysis. Formal Methods in System Design 29(2), 157–175 (2006)
16. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-vincentelli, A.L.: Multi-valued decision diagrams: theory and applications. Multiple-Valued Logic 4(1), 9–62 (1998)
17. Kant, G., Laarman, A.W., Meijer, J., van de Pol, J.C., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015 (2015)
18. Kimura, S., Igaki, T., Haneda, H.: Parallel Binary Decision Diagram Manipulation. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science E75-A(10), 1255–62 (October 1992)
19. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Formal Methods in Computer-Aided Design. pp. 247–255. IEEE (oct 2010)
20. Laarman, A.W., van de Pol, J.C., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. LNCS, vol. 6617, pp. 506–511. Springer (2011)
21. Lovato, A., Macedonio, D., Spoto, F.: A Thread-Safe Library for Binary Decision Diagrams. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings. LNCS, vol. 8702, pp. 35–49. Springer (2014)
22. Miller, D.M., Drechsler, R.: On the Construction of Multiple-Valued Decision Diagrams. In: 32nd IEEE International Symposium on Multiple-Valued Logic (ISMVL 2002). pp. 245–253. IEEE Computer Society (2002)
23. Milvang-Jensen, K., Hu, A.J.: BDDNOW: A parallel BDD package. In: FMCAD. LNCS, vol. 1522, pp. 501–507 (1998)
24. Ossowski, J.: JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation. Ph.D. thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn (October 2010)
25. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN. pp. 263–267. Springer-Verlag, Berlin, Heidelberg (2007)
26. Sahoo, D., Jain, J., Iyer, S.K., Dill, D.L., Emerson, E.A.: Multi-threaded reachability. In: Proceedings of the 42nd annual Design Automation Conference. pp. 467–470. DAC '05, ACM, New York, NY, USA (2005)
27. Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: Proceedings of the 33rd annual Design Automation Conference. pp. 641–644. DAC '96, ACM, New York, NY, USA (1996)