Int J Softw Tools Technol Transfer (2017) 19:675–696
DOI 10.1007/s10009-016-0433-2

TACAS 2015

Sylvan: multi-core framework for decision diagrams

Tom van Dijk[1] · **Jaco van de Pol**[2]

Published online: 19 October 2016

Abstract Decision diagrams, such as binary decision diagrams, multi-terminal binary decision diagrams and multi-valued decision diagrams, play an important role in various fields. They are especially useful to represent the characteristic function of sets of states and transitions in symbolic model checking. Most implementations of decision diagrams do not parallelize the decision diagram operations. As performance gains in the current era now mostly come from parallel processing, an ongoing challenge is to develop datastructures and algorithms for modern multi-core architectures. The decision diagram package Sylvan provides a contribution by implementing parallelized decision diagram operations and thus allowing sequential algorithms that use decision diagrams to exploit the power of multi-core machines. This paper discusses the design and implementation of Sylvan, especially an improvement to the lock-free unique table that uses bit arrays, the concurrent operation cache and the implementation of parallel garbage collection. We extend Sylvan with multi-terminal binary decision diagrams for integers, real numbers and rational numbers. This extension also allows for custom MTBDD leaves and operations and we provide an example implementation of GMP rational numbers. Furthermore, we show how the provided framework can be integrated in existing tools to provide out-of-the-box parallel BDD algorithms, as well as support for the parallelization of higher-level algorithms. As a case study, we parallelize on-the-fly symbolic reachability in the model checking toolset LTSmin. We experimentally demonstrate that the parallelization of symbolic model checking for explicit-state modeling languages, as supported by LTSmin, scales well. We also show that improvements in the design of the unique table result in faster execution of on-the-fly symbolic reachability.

Keywords Multi-core · Parallel · Binary decision diagrams · Multi-terminal binary decision diagrams · Multi-valued decision diagrams · Symbolic model checking

Work funded by the NWO project MaDriD, Grant Nr. 612.001.101.

✉ Tom van Dijk
tom@tvandijk.nl

Jaco van de Pol
J.C.vandePol@utwente.nl

1 Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

2 Formal Methods and Tools, University of Twente, Enschede, The Netherlands

1 Introduction

In model checking, we create models of complex systems to verify that they function according to certain properties. Systems are modeled using possible states and transitions between these states. An important part of many model checking algorithms is state-space exploration using a reachability algorithm, to compute all states reachable from some initial state. A major challenge is that the space and time requirements of these algorithms increase exponentially with the size of the models. One method to alleviate this problem is symbolic model checking [12], where states are not treated individually but as sets of states, stored in binary decision diagrams (BDDs). For many symbolic model checking algorithms, most time is spent in the BDD operations. Another method uses parallel computation, e.g., in computer systems with multiple processors. In [21,23,26], we combined both approaches by parallelizing BDD operations in the parallel BDD library Sylvan.

Contributions This paper is an extended version of [26]. We refer also to the PhD thesis of the first author [21] for a more extensive treatment of multi-core decision diagrams.

In [26], we presented an extension to Sylvan that implements operations on list decision diagrams (LDDs). We also investigated applying parallelism on a higher level than the BDD/LDD operations. Since computing the full transition relation is expensive, the model checking toolset LTSMIN [7,24,38,42] has the notion of transition groups, which disjunctively partition the transition relation. We exploited the fact that partitioned transition relations can be applied in parallel and showed that this results in improved scalability. In addition, LTSMIN supports learning transition relations on-the-fly, which enables the symbolic model checking of explicit-state models. We implemented a specialized operation `collect`, which combines `enumerate` and `union`, to perform parallel transition learning and we showed that this results in good parallel performance.

Since [26], we equipped Sylvan with a versatile implementation of MTBDDs, allowing symbolic computations on integers, floating-points, rational numbers and other types. We discuss the design and implementation of our MTBDD extension, as well as an example of custom MTBDD leaves with the GMP library. Furthermore, we redesigned the unique table to require fewer `cas` operations per created node. We also describe the operation cache and parallel garbage collection in Sylvan.

Experiments on the BEEM database of explicit-state models show that parallelizing the higher level algorithms in LTSMIN pays off, as the parallel speedup increases from $5.6\times$ to $16\times$, while the sequential computation time (with 1 worker) stays the same. The experiments also show that LDDs perform better than BDDs for this set of benchmarks. In addition to the experiment performed in [26], we include additional experiments using the new hash table. These benchmark results show that the new hash table results in a 21 % faster execution for 1 worker, and a 30 % faster execution with 48 workers, improving the parallel speedup from $16\times$ to $18\times$.

Outline This paper is organized as follows. After a review of the related work in Sect. 2, we introduce decision diagrams and parallel programming in Sect. 3. Section 4 discusses how we use work-stealing to parallelize operations. Section 5 presents the implementation of the datastructures of the unique table and the operation cache, as well as the implementation of parallel garbage collection in Sylvan. Section 6 discusses the implementation of specific decision diagram operations, especially the BDD and MTBDD operations. In Sect. 7, we apply parallelization to on-the-fly symbolic reachability in LTSMIN. Section 8 shows the results of several experiments using the BEEM database of explicit-state models to measure the effectiveness of our approach. Finally, Sect. 9 summarizes our findings and reflections.

2 Related work

This section is largely based on earlier literature reviews we presented in [23,26].

Massively parallel computing (early '90s) In the early '90s, researchers tried to speed up BDD manipulation by parallel processing. The first paper [39] views BDDs as automata, and combines them by computing a product automaton followed by minimization. Parallelism arises by handling independent subformulae in parallel: the expansion and reduction algorithms themselves are not parallelized. They use locks to protect the global hash table, but this still results in a speedup that is almost linear with the number of processors. Most other work in this era implemented BFS algorithms for vector machines [47] or massively parallel SIMD machines [13,32] with up to 64K processors. Experiments were run on supercomputers, like the Connection Machine. Given the large number of processors, the speedup (around 10–20) was disappointing.

Parallel operations and constructions An interesting contribution in this period is the paper by Kimura et al. [40]. Although they focus on the construction of BDDs, their approach relies on the observation that suboperations from a logic operation can be executed in parallel and the results can be merged to obtain the result of the original operation. Our solution to parallelizing BDD operations follows the same line of thought, although the work-stealing method for efficient load balancing that we use was first published 2 years later [8]. Similar to [40], Parasuram et al. implement parallel BDD operations for distributed systems, using a "distributed stack" for load balancing, with speedups from 20–32 on a CM-5 machine [50]. Chen and Banerjee implemented the parallel construction of BDDs for logic circuits using lock-based distributed hash tables, parallelizing on the structure of the circuits [14]. Yang and O'Hallaron [60] parallelized breadth-first BDD construction on multi-processor systems, resulting in reasonable speedups of up to $4\times$ with 8 processors, although there is a significant synchronization cost due to their lock-protected unique table.

Distributed memory solutions (late '90s) Attention shifted towards Networks of Workstations, based on message passing libraries. The motivation was to combine the collective memory of computers connected via a fast network. Both depth-first [3,5,57] and breadth-first [53] traversal have been proposed. In the latter, BDDs are distributed according to variable levels. A worker can only proceed when its level has a turn, so these algorithms are inherently sequential. The advantage of distributed memory is not that multiple machines can perform operations faster than a single machines, but that their memory can be combined in to handle larger BDDs. For example, even though [57] reports a nice parallel speedup, the performance with 32 machines is still $2\times$ slower than the non-parallel version. BDDNOW [46] is the first BDD package that reports some speedup compared to the non-parallel version, but it is still very limited.

Parallel symbolic reachability (after 2000) After 2000, research attention shifted from parallel implementations of BDD operations towards the use of BDDs for symbolic reachability in distributed [15,33] or shared memory [18,28]. Here, BDD partitioning strategies such as horizontal slicing [15] and vertical slicing [35] were used to distribute the BDDs over the different computers. Also, the saturation algorithm [16], an optimal iteration strategy in symbolic reachability, was parallelized using horizontal slicing [15] and using the work-stealer Cilk [28], although it is still difficult to obtain good parallel speedup [18].

Multi-core BDD algorithms There is some recent research on multi-core BDD algorithms. There are several implementations that are thread-safe, i.e., they allow multiple threads to use BDD operations in parallel, but they do not offer parallelized operations. In a thesis on the BDD library JINC [49], Chapter 6 describes a multi-threaded extension. JINC's parallelism relies on concurrent tables and delayed evaluation. It does not parallelize the basic BDD operations, although this is mentioned as possible future research. Also, a recent BDD implementation in Java called BeeDeeDee [43] allows execution of BDD operations from multiple threads, but does not parallelize single BDD operations. Similarly, the well-known sequential BDD implementation CUDD [56] supports multi-threaded applications, but only if each thread uses a different "manager", i.e., unique table to store the nodes in. Except for our contributions [23,24,26] related to Sylvan, there is no recent published research on modern multi-core shared-memory architectures that parallelizes the actual operations on BDDs. Recently, Oortwijn [48] continued our work by parallelizing BDD operations on shared memory abstractions of distributed systems using remote direct memory access. Also, Velev and Gao [58] have implemented parallel BDD operations on a GPU using a parallel cuckoo hash table.

Finally, we refer to Somenzi [55] for a detailed paper on the implementation of decision diagrams, and to the PhD thesis of the first author [21] on multi-core decision diagrams.

3 Preliminaries

This section presents the definitions of binary decision diagrams (BDDs), multi-valued decision diagrams (MDDs), multi-terminal binary decision diagrams (MTBDDs) and list decision diagrams (LDDs) from the literature [4,6,11,37]. Furthermore, we discuss parallel programming.

3.1 Decision diagrams

Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions $\mathbb{B}^N \to \mathbb{B}$ [2,11]. They are a basic structure in discrete mathematics and computer science. A (reduced, ordered) BDD is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1, called the "low" and the "high" edge. Furthermore, variables are encountered along each directed path according to a fixed variable ordering. Duplicate nodes (two nodes with the same variable label and outgoing edges) and nodes with two identical outgoing edges (redundant nodes) are forbidden. It is well known that, given a fixed variable ordering, every Boolean function is represented by a unique BDD [11].

In addition, we use complement edges [10] as a property of an edge to denote the negation of a BDD, i.e., the leaf 1 is interpreted as 0 and vice versa, or in general, each leaf is interpreted as its negation. This is a well-known technique. We write $\neg$ to denote toggling this property on an edge. BDDs with complement edges require an extra rule to remain canonical representations of Boolean functions: the complement mark must be forbidden on either the high or the low edges. We choose to forbid complement edges on the low edges. BDDs with complement edges are interpreted as follows: if the high edge has a complement mark, then the BDD node represents the Boolean function $x\neg f_{x=1} \vee \overline{x} f_{x=0}$, otherwise $x f_{x=1} \vee \overline{x} f_{x=0}$, where $f_{x=1}$ and $f_{x=0}$ are computed by interpreting the BDDs obtained by following the high and the low edges. See Fig. 1 for several examples of simple BDDs, with and without the use of complement edges.

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams (MTBDDs) have been proposed [4,19] with arbitrary leaves, representing functions from the Boolean space $\mathbb{B}^N$ onto any set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), real numbers (encoding $\mathbb{B}^N \to \mathbb{R}$) and rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). In our implementation of MTBDDs, we also allow for partially defined functions, using a leaf $\bot$. See Fig. 2 for an example of an MTBDD.

Multi-valued decision diagrams (MDDs, sometimes also called multi-way decision diagrams) are a generalization of BDDs to the other domains, such as integers [37]. Whereas BDDs represent functions $\mathbb{B}^N \to \mathbb{B}$, MDDs represent functions $\mathbb{D}_1 \times \cdots \times \mathbb{D}_N \to \mathbb{B}$, for finite domains $\mathbb{D}_1, \ldots, \mathbb{D}_N$. They are typically used to represent functions on integer domains like $(\mathbb{N}_{<v})^N$. Rather than two outgoing edges, each internal MDD node with variable x_i has n_i labeled outgoing edges. For example for integers, these edges could be labeled

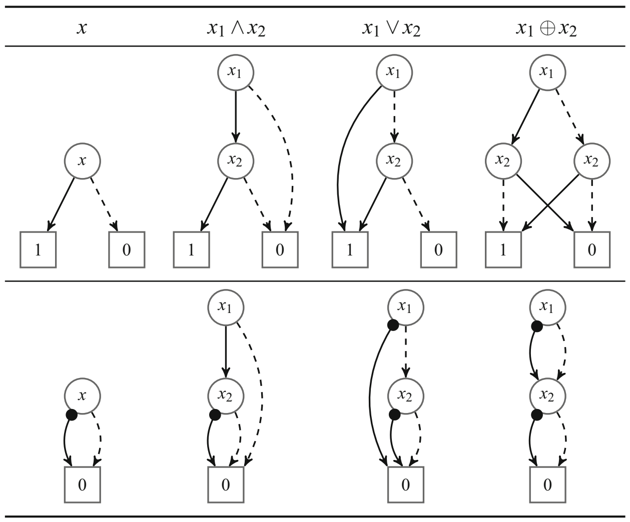


Fig. 1 Binary decision diagrams for several Boolean functions, without complement edges (*above*) and with complement edges (*below*). Internal nodes are drawn as *circles* with variables, and leaves as *boxes*. High edges are drawn *solid*, and low edges are drawn *dashed*. BDDs are evaluated by following the high edge when a variable x is `true`, or the low edge when it is `false`

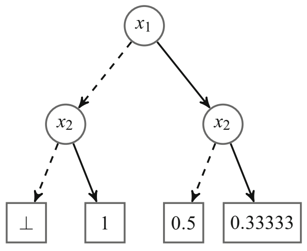


Fig. 2 The MTBDD for a function that maps $\overline{x_1}x_2$ to 1, $x_1\overline{x_2}$ to 0.5, and x_1x_2 to 0.33333. The function is undefined for the input $\overline{x_1}\,\overline{x_2}$

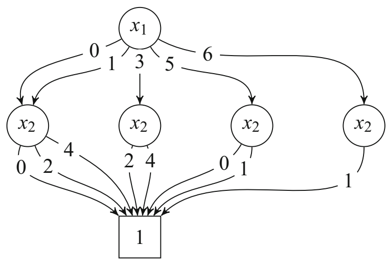


Fig. 3 Edge-labeled MDD (hiding paths to 0) for the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$

0 to $n_i - 1$. See Fig. 3 for an MDD representing a set of integer pairs, where we hide edges to 0 to improve the readability.

As an alternative to MDDs, list decision diagrams (LDDs) represent sets of integer vectors, such as sets of states in model checking. List decision diagrams encode functions $(\mathbb{N}_{<v})^N \to \mathbb{B}$, and were initially described in [6, Sect. 5]. A list decision diagram is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a value v and two outgoing edges labeled $>$ and $=$, also called the "right" and the "down" edge. Along the "right" edges, values v are encountered in ascending order. The "down" edge never points to leaf 0 and the "right" edge never points to leaf 1. Duplicate nodes are forbidden. See Fig. 4 for an example of an LDD that represents the same set as the MDD in Fig. 3.

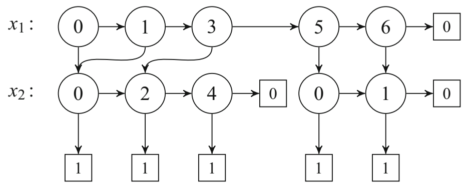


Fig. 4 LDD representing the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$. We draw the same leaf multiple times for aesthetic reasons

LDD nodes have a property called a level (and its dual, depth), which is defined as follows: the root node is at the first level, nodes along "right" edges stay in the same level, while "down" edges lead to the next level. The depth of an LDD node is the number of "down" edges to leaf 1.

LDDs compared to MDDs A typical method to store MDDs in memory stores the variable label x_i plus an array holding all n_i edges (pointers to nodes), e.g., in [45]: `struct node { int lvl; node* edges[]; }`. New nodes are allocated dynamically using `malloc` and a hash table ensures that no duplicate MDD nodes are created. 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 [17], the edges are stored in a separate `int[]` array to allow the number of edges n_i to vary. 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, and outgoing edges to 0 are not stored) using arrays is a waste of memory. (2) MDD nodes typically have a variable size, complicating memory management.

List decision diagrams can be understood as a linked-list representation of "quasi-reduced" MDDs. Quasi-reduced MDDs are a variation of normal (fully-reduced) MDDs. Instead of forbidding redundant nodes (with identical outgoing edges), quasi-reduced MDDs forbid skipping levels. They are canonical representations, like fully-reduced MDDs. An advantage of quasi-reduced MDDs is that, for some applications, edges that do not skip levels can be easier to manage [17]. Also, variables labels do not need to be stored as they follow implicitly from the depth of the MDD.

LDDs have several advantages compared to MDDs [6]. LDD nodes are binary, so they have a fixed node size which is easier for memory allocation. They are better for sparse

```
1  def apply($x$, $y$, F):
2      if $x$ and $y$ are leaves or trivial : return F($x$, $y$)
3      normalize/simplify parameters
4      if result ← cache[($x$, $y$, F)] : return result
5      $v$ = topvar($x$,$y$)
6      low ← apply($x_{v=0}$, $y_{v=0}$, F)
7      high ← apply($x_{v=1}$, $y_{v=1}$, F)
8      result ← lookupBDDnode($v$, low, high)
9      cache[($x$, $y$, F)] ← result
10     return result
```

Algorithm 1 Example of a parallelized BDD algorithm: apply a binary operator F to BDDs x and y

```
1  def lookupBDDnode($x$, low, high):
2      if low = high : return low
3      if complement(low) :
4          return ¬lookupBDDnode($x$, ¬low, ¬high)
5      try :
6          return find-or-insert ({$x$, low, high})
7      catch TableFull :
8          garbage-collect()
9          return find-or-insert ({$x$, low, high})
```

Algorithm 2 The BDDnode method creates a BDD node using the hash table find-or-insert method (Algorithm 3) to ensure that there are no duplicate nodes. Line 2 ensures that there are no redundant nodes

sets: valuations that lead to 0 simply do not appear in the LDD. LDDs also have more opportunities for the sharing of nodes, as demonstrated in the example of Fig. 4, where the LDD encoding the set $\{2, 4\}$ is used for the set $\{0, 2, 4\}$ and reused for the set $\{\langle 3, 2\rangle, \langle 3, 4\rangle\}$, and similarly, the LDD encoding $\{1\}$ is used for $\{0, 1\}$ and for $\{\langle 6, 1\rangle\}$. A disadvantage of LDDs is that their linked-list style introduces edges "inside" the MDD nodes, requiring more memory pointers, similar to linked lists compared with arrays.

3.2 Decision diagram operations

Operations on decision diagrams are typically recursively defined. Suboperations are computed based on the subgraphs of the inputs, i.e., the decision diagrams obtained by following the two outgoing edges of the root node, and their results are used to compute the result of the full operation. In this subsection we look at Algorithm 1, a generic example of a BDD operation. This algorithm takes as inputs the BDDs x and y (with the same fixed variable ordering), to which a binary operation F is applied. We assume that, given the same parameters, F always returns the same result. Therefore, we use a cache to store the results of (sub)operations. This is in fact required to reduce the complexity class of many BDD operations from exponential time to polynomial time.

Most decision diagram operations first check if the operation can be applied immediately to x and y (line 2). This is typically the case when x and y are leaves. Often there are also other trivial cases that can be checked first. After this, the operation cache is consulted (lines 3–4). In cases where computing the result for leaves or other cases takes a significant amount of time, the cache should be consulted first. Often, the parameters can be normalized in some way to increase the cache efficiency. For example, $a \wedge b$ and $b \wedge a$ are the same operation. In that case, normalization rules can rewrite the parameters to some standard form to increase cache utilization, at line 3. A well-known example is the if-then-else algorithm, which rewrites using rewrite rules called "standard triples" as described in [10].

If x and y are not leaves and the operation is not trivial or in the cache, we use a function topvar (line 5) to determine the first variable of the root nodes of x and y. If x and y have a different variable in their root node, topvar returns the first one in the variable ordering of x and y. We then compute the recursive application of F to the cofactors of x and y with respect to the variable v in lines 6–7. We write $x_{v=i}$ to denote the cofactor of x where variable v takes value i. Since x and y are ordered according to the same fixed variable ordering, we can easily obtain $x_{v=i}$. If the root node of x is on the variable v, then $x_{v=i}$ is obtained by following the low ($i = 0$) or high ($i = 1$) edge of x. Otherwise, $x_{v=i}$ equals x. After computing the suboperations, we compute the result by either reusing an existing or creating a new BDD node (line 8). This is done by the operation lookupBDDnode which, given a variable v and the BDDs of $\text{result}_{v=0}$ and $\text{result}_{v=1}$, returns the BDD for result. Finally, the result is stored in the cache (line 9) and returned (line 10).

The operation lookupBDDnode is given in Algorithm 2. This operation ensures that there are no redundant nodes (line 2) and no complement mark on the low edge (lines 3–4) and employs the method find-or-insert (implemented by the unique table, see Sect. 5) to ensure that there are no duplicate nodes (lines 6 and 9). If the hash table is full, then garbage collection is performed (line 8).

3.3 Parallel programming

In parallel programs, memory accesses can result in race conditions or data corruption, for example when multiple threads write to the same memory location. Often datastructures are protected against race conditions using locking techniques. While locks are relatively easy to implement and reason about, they can severely cripple parallel performance, especially as the number of threads increases. Threads must wait until the lock is released, and locks can be a bottleneck when many threads try to acquire the same lock. Also, locks can sometimes cause spurious delays that smarter datastructures could avoid, for example by recognizing that some operations do not interfere even though they access the same resource.

A standard technique that avoids locks uses the atomic `compare-and-swap(cas)` operation, which is supported by many modern processors.

```
1 def compare-and-swap(address, expected, newval):
2     value ← *address
3     if value $\neq$ expected : return False
4     *address ← newval
5     return True
```

This operation atomically compares the contents of a given location in shared memory to some given expected value and, if the contents match, changes the contents to a given new value. If multiple processors try to change the same bytes in memory using `cas` at the same time, then only one succeeds.

Datastructures that avoid locks are called non-blocking or lock-free. Such datastructures often use the atomic `cas` operation to make progress in an algorithm, rather than protecting a part that makes progress. For example, when modifying a shared variable, an approach using locks would first acquire the lock, then modify the variable, and finally release the lock. A lock-free approach would use atomic `cas` to modify the variable directly. This requires only one memory write rather than three, but lock-free approaches are typically more complicated to reason about, and prone to bugs that are more difficult to reproduce and debug.

There is a distinction between different levels of lock freedom. We are concerned with three levels:

– In **blocking** datastructures, it may be possible that no threads make progress if a thread is suspended. If an operation may be delayed forever because another thread is suspended, then that operation is blocking.
– In **lock-free** datastructures, if any thread working on the datastructure is suspended, then other threads must still be able to perform their operations. An operation may be delayed forever, but if this is because another thread is making progress and never because another thread is suspended, then that operation is lock-free.
– In **wait-free** datastructures, every thread can complete its operation within a bounded number of steps, regardless of the other threads; all threads make progress.

3.4 System architecture

This paper assumes a cache coherent shared memory NUMA architecture, i.e., there are multiple processors and multiple memories, with a hierarchy of caches, all connected via interconnect channels. The shared memory is divided into regions called cachelines, which are typically 64 bytes long. Only whole cachelines are communicated between processors and with the memory. Datastructures designed for multi-core shared-memory architectures should aim to minimize the number of cacheline transfers to be efficient. We also assume the x86 TSO memory model [54]. In this memory model, memory writes of each processor are not reordered, but memory writes can be buffered. The datastructures presented in this paper rely on `compare-and-swap` instructions and assume total store ordering for their correctness.

4 Parallel operations using work-stealing

This section describes how we use work-stealing to execute operations on decision diagrams in parallel.

We implement recursively defined operations such as Algorithm 1 as independent tasks using a task-based parallel framework. For task parallelism that fits a “strict” fork-join model, i.e., each task creates the subtasks that it depends on, work-stealing is well known to be an effective load balancing method [8], with implementations such as Cilk [9,31] and Wool [29,30] that allow writing parallel programs in a style similar to sequential programs [1]. Work-stealing has been proven to be optimal for a large class of problems and has tight memory and communication bounds [8].

In work-stealing, tasks are executed by a fixed number of workers, typically equal to the number of processor cores. Each worker owns a task pool into which it inserts new subtasks created by the task it currently executes. Idle workers steal tasks from the task pools of other workers. Workers are idle either because they do not have any tasks to perform (e.g., at the start of a computation), or because all their tasks have been stolen and they have to wait for the result of the stolen tasks to continue the current task. Typically, one worker starts executing a root task and the other workers perform work-stealing to acquire subtasks.

We use **do in parallel** to denote that tasks are executed in parallel. Programs in the Cilk/Wool style are then implemented like in Fig. 5. The `SPAWN` keyword creates a new task. The `SYNC` keyword matches with the last unmatched `SPAWN`, i.e., operating as if spawned tasks are stored on a stack. It waits until that task is completed and retrieves the result. Every `SPAWN` during the execution of the program must have a matching `SYNC`. The `CALL` keyword skips the task stack and immediately executes a task.

Decision diagram operations like Algorithm 1 are parallelized by executing lines 6–7 in parallel:

```
1 do in parallel:
2     K ← F1(x, y, z)
3     L ← F2(a, b, c)
4     M ← F3(g, h)
```

```
1 SPAWN(F1, x, y, z)
2 SPAWN(F2, a, b, c)
3 M ← CALL(F3, g, h)
4 L ← SYNC
5 K ← SYNC
```

Fig. 5 The algorithm (*left*) is implemented (*right*) using `SPAWN`, `SYNC` and `CALL`

```
6 do in parallel:
7     low ← apply($x_{v=0}$, $y_{v=0}$, F)
8     high ← apply($x_{v=1}$, $y_{v=1}$, F)
```

This is equivalent to the following:

```
6 SPAWN(apply, $x_{v=0}$, $y_{v=0}$, F)
7 high ← CALL(apply, $x_{v=1}$, $y_{v=1}$, F)
8 low ← SYNC
```

We substituted the work-stealing framework Wool [29], that we used in the first published version of Sylvan [23], by Lace [25], which we developed based on ideas to minimize interactions between workers and with the shared memory. Lace is based around a novel work-stealing queue, which is described in detail in [25]. Lace also implements extra features necessary for parallel garbage collection.

To implement tasks, Lace provides C macros that require only few modifications of the original source code. One helpful feature for garbage collection in Sylvan that we implemented in Lace is a feature that suspends all current tasks and starts a new task tree. Lace implements a macro `NEWFRAME(...)` that starts a new task tree, where one worker executes the given task and all other workers perform work-stealing to help execute this task in parallel. The macro `TOGETHER(...)` also starts a new task tree, but all workers execute a local copy of the given task.

Sylvan uses the `NEWFRAME` macro as part of garbage collection, and the `TOGETHER` macro to perform thread-specific initialization. Programs that use Sylvan can also use the Lace framework to parallelize their high-level algorithms. We give an example of this in Sect. 7.

5 Concurrent datastructures

This section describes the concurrent datastructures required to parallelize decision diagram operations. Every operation requires a scalable concurrent unique table for the BDD nodes and a scalable concurrent operation cache. We use a single unique table for all BDD nodes and a single operation cache for all operations.

The parallel efficiency of a task-based parallelized algorithm depends largely on the contents of each task. For example, tasks that perform many processor calculations and few memory operations typically result in good speedups. Also, tasks that have many subtasks provide load balancing frameworks with ample opportunity to execute independent tasks in parallel. 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 operations typically perform few calculations and are memory-intensive, since they consist mainly of calls to the operation cache and the unique table. Furthermore, BDD operations typically spawn only one or two independent subtasks for parallel execution, depending on the inputs and the operation. Hence the design of scalable concurrent datastructures (for the cache and the unique table) is crucial for the parallel performance of BDD implementation.

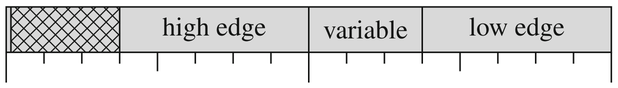
5.1 Representation of nodes

This subsection discusses how BDD nodes, LDD nodes and MTBDD nodes are represented in memory. We use 16 bytes for all types of nodes, so we can use the same unique table for all nodes and have a fixed node size. As we see below, not all bits are needed; unused bits are set to 0. Also, with 16 bytes per node, this means that 4 nodes fit exactly in a cacheline of 64 bytes (the size of the cacheline for many current computer architectures, in particular the x86 family that we use), which is very important for performance. If the unique table is properly aligned in memory, then only one cacheline needs to be accessed when accessing a node.

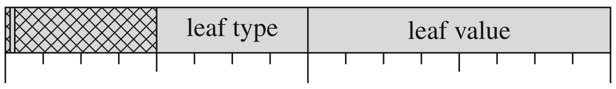
We use 40 bits to store the index of a node in the unique table. This is sufficient to store up to 2^{40} nodes, i.e. 16 terabytes of nodes, excluding overhead in the hash table (to store all the hashes) and other datastructures. As we see below, there is sufficient space in the nodes to increase this to 48 bits per node (up to 4096 terabytes), although that would have implications for the performance (more difficult bit operations) and for the design of the operation cache.

Edges to nodes Sylvan defines the type BDD as a 64-bit integer, representing an edge to a BDD node. The lowest 40 bits represent the location of the BDD node in the nodes table, and the highest-significant bit stores the complement mark [10]. The BDD 0 is reserved for the leaf `false`, with the complemented edge to 0 (i.e. `0x8000000000000000`) meaning `true`. We use the same method for MTBDDs and LDDs, although most MTBDDs do not have complemented edges. LDDs do not have complemented edges at all. The LDD leaf `false` is represented as 0, and the LDD leaf `true` is represented as 1. For the MTBDD leaf ⊥ we use the leaf 0 that represents Boolean `false` as well. This has the advantage that Boolean MTBDDs can act as filters for MTBDDs with the MTBDD operation `times`. The disadvantage is that partial Boolean MTBDDs are not supported by default, but can easily be implemented using a custom MTBDD leaf.

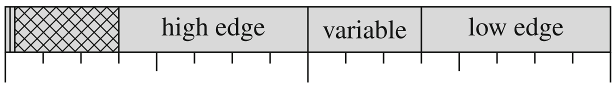
Internal BDD nodes Internal BDD nodes store the variable label (24 bits), the low edge (40 bits), the high edge (40 bits), and the complement bit of the high edge (the first bit below).



MTBDD leaves For MTBDDs we use a bit that indicates whether a node is a leaf or not. MTBDD leaves store the leaf type (32 bits), the leaf contents (64 bits) and the fact that they are a leaf (1 bit, set to 1):

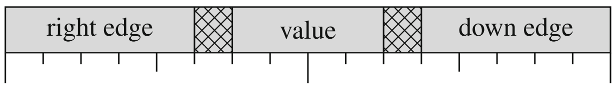


Internal MTBDD nodes Internal MTBDD nodes store the variable label (24 bits), the low edge (40 bits), the high edge (40 bits), the complement bit of the high edge (1 bit, the first bit below) and the fact they are not a leaf (1 bit, the second bit below, set to 0).



Internal BDD nodes are identical to internal MTBDD nodes, as unused bits are set to 0. Hence, the BDD 0 can be used as a terminal for Boolean MTBDDs, and the resulting Boolean MTBDD is identical to a BDD of the same function.

Internal LDD nodes Internal LDD nodes store the value (32 bits), the down edge (40 bits) and the right edge (40 bits):



5.2 Scalable unique table

This subsection describes the hash tables that we use to store the unique decision diagram nodes. We refer to [21] for a more extensive treatment of these hash tables.

The hash tables store fixed-size decision diagram nodes (16 bytes for each node) and strictly separate lookup and insertion of nodes from a stop-the-world garbage collection phase, during which the table may be resized. From the perspective of the nodes table algorithms (and correctness), all threads of the program are in one of two phases:

1. During **normal operation**, threads only call the method `find-or-insert`, which takes as input the node and either returns a unique identifier for the data, or raises the TableFull signal if the algorithm fails to insert the data.
2. During **garbage collection**, `find-or-insert` is never called.

This simplifies the requirements for the hash tables. The `find-or-insert` operation must have the following property: if the operation returns a value for some given data, then other `find-or-insert` operations may not return

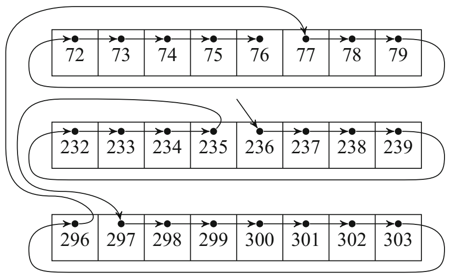


Fig. 6 Example of the walking-the-line probe sequence, with the starting buckets *236*, *297* and *77* based on the first three hash values of the data

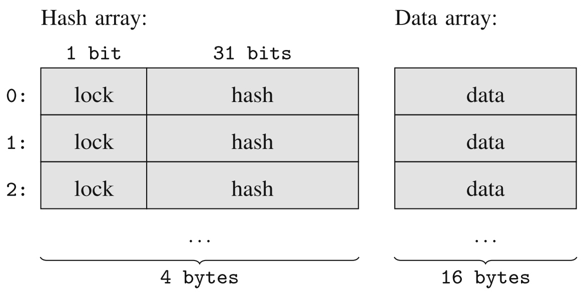


Fig. 7 Layout of the hash table in [41] using a separate hash array and data array

the same value for a different input, or return a different value for the same input. This property must hold between garbage collections; garbage collection obviously breaks the property for nodes that are not kept during garbage collection, as nodes are removed from the table to make room for new data.

In [26], we implemented a hash table based on the lockless hash table presented in [41]. The datastructures in [41] and [26] are based on the following ideas:

– Using a probe sequence called “walking-the-line” that is efficient with respect to transferred cachelines. See also Fig. 6.
– Using a light-weight parametrised local “writing lock” when inserting data, which almost always only delays threads that insert the same data.
– Separating the stored data in a “data array” and the hash of the data in the “hash array” so directly comparing the stored data is often avoided. See also Fig. 7.

Probe sequence Every hash table needs to implement a strategy to deal with hash table collisions, i.e., when different data hashes to the same location in the table. To find a location for the data in the hash table, some hash tables use open addressing: they visit buckets in the hash table in a deterministic order called the probe sequence. One of the simplest probe sequences is linear probing, where the data is hashed once to obtain the first bucket (e.g., bucket 61), and the probe sequence consists of all buckets from that first bucket (e.g., 61, 62, 63...). An alternative to linear probing is walking-the-line, proposed in [41]. Since data in a computer is transferred in blocks called cachelines, it is more efficient to use the entire cacheline instead of only a part of the cacheline. Walking-the-line is similar to linear probing, but continues at the beginning of the cacheline when the end has been reached. After the whole cacheline has been checked, a new hash value is computed for the next bucket. See Fig. 6 for an example of walking-the-line.

Writing lock When multiple workers simultaneously access the hash table to find or insert data, there must be some mechanism to avoid race conditions, such as inserting the same data twice, or trying to insert different data at the same location simultaneously. Rather than using a global lock on the entire hash table or regions of the hash table, or a non-specific local lock on each bucket, the hash table of [41] combines a short-lived local lock with a hash value of the data that is inserted. This way, threads that are finding or inserting data with a different hash value know that they can skip the locked bucket in their search.

An empty bucket is first locked using an atomic `cas` operation that sets the lock with the hash value of the inserted data, then writes the data, and then releases the lock. Only workers that are finding or inserting data with the same hash as the locked bucket need to wait until the lock is released. This approach is not lock-free. The authors state that a mechanism could be implemented that ensures local progress (making the algorithm wait-free), however, this is not needed, since the writing locks are rarely hit under normal operation [41].

Separated arrays The hash table stores the hash of the data and the short-lived lock separated from the stored data. The idea is that the `find-or-insert` algorithm does not need to access the stored data if the stored hash does not match with the hash of the data given to `find-or-insert`. This reduces the number of accessed cachelines during `find-or-insert`. See also Fig. 7. Each bucket i in the hash array matches with the bucket i in the data array. The hash that is stored in the hash array is independent of the hash value used to determine the starting bucket in the probe sequence, although in practice hash functions give a 64-bit or 128-bit hash that we can use both to determine the starting bucket in the probe sequence and the 31-bit hash for the hash array.

The hash table presented in [26] stores independent locations for the bucket in the hash array and in the data array. The idea is that the location of the decision diagram node in the data array is used for the node identifier and that nodes can be reinserted into the hash array without changing the node identifier. This is important, since garbage collection is performed often and nodes identifiers should remain unchanged

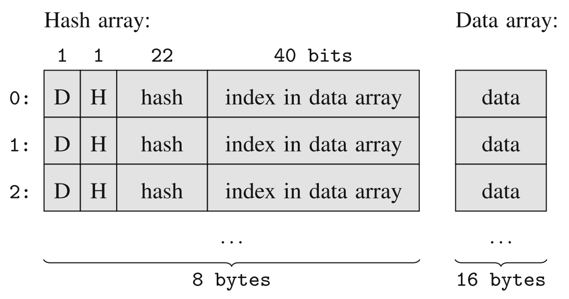


Fig. 8 Layout of the hash table [26] with hash array `h` and data array `d`. The field D of hash bucket i controls whether the data bucket i is used; the field H of hash bucket i controls whether the hash bucket i is used, i.e., the fields hash and index

during garbage collection, i.e., nodes should not be moved. To implement this feature, the buckets from the hash array are extended to contain the index in the data array where the corresponding data is stored, as well as a bit that controls whether the bucket in the data array with the same index is in use (see Fig. 8). See further [26].

In this paper, we present a redesigned version of the hash table that uses bit arrays to control access to the data array.

The hash table in [26] has the drawback that the speculative insertion and uninsertion into the data array requires atomic `cas` operations, once for the insertion, once for the uninsertion. Instead of using a field D in the hash array, we use a separate bit array `databits` to implement a parallel allocator for the data array. Furthermore, to avoid having to use `cas` for every change to `databits`, we divide this bit array into regions, such that every region matches exactly with one cacheline of the `databits` array, i.e., 512 buckets per region if there are 64 bytes in a cacheline, which is the case for most current architectures. Every worker has exclusive access to one region, which is managed with a second bit array `regionbits`. Only changes to `regionbits` (to claim a new region) require an atomic `cas`. The new version therefore, only uses normal writes for insertion and uninsertion into the data array, and only occasionally an atomic `cas` during speculative insertion to obtain exclusive access to the next region of 512 buckets.

A claimed region is not given back until garbage collection, which resets claimed regions. On startup and after garbage collection, the `regionbits` array is cleared and all threads claim a region using the `claim-next-region` method in Algorithm 3. All threads start at a different position (distributed over the entire table) for their first claimed region, to minimize the interactions between threads. The `databits` array is empty at startup and during garbage collection threads use atomic `cas` to set the bits in `databits` of decision diagram nodes that must be kept in the table. In addition, the bit of the first bucket is always set to 1 to avoid using the index 0 since this is a reserved value in Sylvan.

```
1  def find-or-insert(data):
2      index ← 0
3      h ← hash(data)
4      for s ∈ probe-sequence(data) :
5          V ← harray[s]
6          if $V = 0$ :
7              if index = 0 :
8                  index ← reserve-data-bucket()
9                  darray[index] ← data
10             if cas(harray[s], 0, {h, index}) : return index
11             else: V ← harray[s]
12         if V.hash = h ∧ darray[V.index] = data :
13             if index ≠ 0 : free-data-bucket(index)
14             return V.index
15     raise TableFull
16 def reserve-data-bucket():
17     loop:
18         if myregion has a bit set to 0 :
19             i ← first bit in myregion that is 0
20             set-bit(databits, 512 × myregion + i, 1)
21             return 512 × myregion + i
22         else: myregion ← claim-next-region(myregion)
23 def free-data-bucket(d):
24     set-bit(databits, d, 0)
25 def claim-next-region(oldregion):
26     newregion ← (oldregion + 1) mod (tablesize/512)
27     while newregion ≠ oldregion :
28         loop:
29             if the bit for newregion is 1 : break
30             if set-bit-cas(regionbits, newregion, 0, 1) :
               return newregion
31         newregion ← (newregion + 1) mod (tablesize/512)
32     raise TableFull
```

Algorithm 3 Algorithm for parallel `find-or-insert` of the hash table, with 512 buckets per region. The variable `myregion` is a thread-specific variable

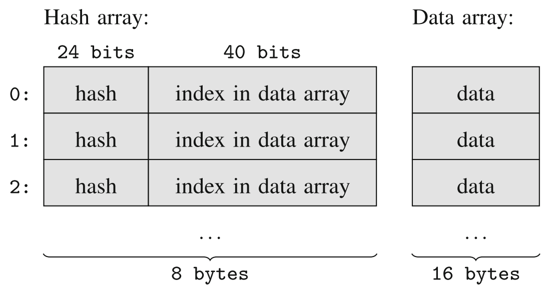


Fig. 9 Layout of the hash array and data array in the new hash table design

The layout of the hash array and the data array is given in Fig. 9. We also remove the field H, which is obsolete as we use a hash function that never hashes to 0 and we forbid nodes with the index 0 because 0 is a reserved value in Sylvan. The fields hash and index are therefore, never 0, unless the hash bucket is `empty`, so the field H to indicate that hash and index have valid values is not necessary. Manipulating the hash array bucket is also simpler, since we no longer need to take into account changes to the field D.

Inserting data into the hash table consists of three steps. First the algorithm determines whether the data is already in the table. If this is not the case, then a new bucket in the data array is reserved in the current region of the thread with `reserve-data-bucket`. If the current region is full, then the thread claims a new region with `claim-next-region`. It may be possible that the next region contains used buckets, if there has been a garbage collection earlier, or even that it is already full for this reason. When the data has been inserted into an available bucket in the the data array, the (hash and index of) the data is also inserted into the hash array. Sometimes, the data has been inserted concurrently (by another thread) and then the bucket in the data array is freed again with the `free-data-bucket` function, so it is available the next time the thread wants to insert data.

The main method of the hash table is `find-or-insert`. See Algorithm 3. The algorithm uses the local variable "index" to keep track of whether the data is inserted into the data array. This variable is initialized to 0 (line 2) which signifies that data is not yet inserted in the data array. For every bucket in the probe sequence, we first check if the bucket is empty (line 6). In that case, the data is not yet in the table. If we did not yet write the data in the data array, then we reserve the next bucket and write the data (lines 7–9). We use atomic `cas` to insert the hash and index into the hash array (line 10). If this is successful, then the algorithm is done and returns the location of the data in the data array. If the `cas` operation fails, some other thread inserted data here and we refresh our knowledge of the bucket (line 11) and continue at line 12. If the bucket is not or no longer empty, then we compare the stored hash with the hash of our data, and if this matches, we compare the data in the data array with the given input (line 12). If this matches, then we may need to free the reserved bucket (line 13) and we return the index of the data in the data array (line 14). If we finish the probe sequence without inserting the data, we raise the TableFull signal (line 15).

The `find-or-insert` method relies on the methods `reserve-data-bucket` and `free-data-bucket`, which are also given in Algorithm 3. They are straightforward.

The `claim-next-region` method searches for the first 0-bit in the `regionbits` array. The value `tablesize` here represents the size of the entire table. We use a simple linear search and a `cas`-loop to actually claim the region. Note that we may be competing with threads that are trying to set the bit of a different region, since the smallest range for the atomic `cas` operation is 1 byte or 8 bits.

The algorithms in Algorithm 3 are wait-free. The method `claim-next-region` is wait-free, since the number of `cas` failures is bounded: regions are only claimed and not released (until garbage collection), and the number of regions is bounded, so the maximum number of cas failures is the number of regions. The `free-data-bucket` is trivially wait-free: there are no loops. The `reserve-data-bucket` method contains a loop, but since `claim-next-region` is wait-free and the number of times `claim-next-region` returns a value instead of raising the TableFull signal is bounded by the number of regions, `reserve-data-bucket` is also wait-free. Finally the `find-or-insert` method only relies on wait-free methods and has only one for-loop (line 4) which is bounded by the number of items in the probe sequence. It is therefore, also wait-free.

5.3 Scalable operation cache

The operation cache is a hash table that stores intermediate results of BDD operations. It is well known that an operation cache is required to reduce the worst-case time complexity of BDD operations from exponential time to polynomial time.

In practice, we do not guarantee this property. Since Sylvan is a parallel package, it is possible that multiple workers compute the same operation simultaneously. While operations could use the operation cache to "claim" a computation (using a dummy result and promising a real result later), we found that the amount of duplicate work due to parallelism is limited. In addition, to guarantee polynomial time, the operation cache must store every subresult. In practice, we find that we obtain a better performance by caching only *many* results instead of all results, and by allowing the cache to *overwrite* earlier results when there is a hash collision.

In [55], Somenzi writes that a lossless computed table guarantees polynomial cost for the basic synthesis operations, but that lossless tables (that do not throw away results) are not feasible when manipulating many large BDDs and in practice lossy computed tables (that may throw away results) are implemented. If the cost of recomputing subresults is sufficiently small, it can pay off to regularly delete results or even prefer to sometimes skip the cache to avoid data races. We design our operation cache below to abort operations as fast as possible when there may be a data race or the data may already be in the cache.

On top of this, our BDD implementation implements *caching granularity*, which controls when results are cached. Most BDD operations compute a result on a variable x_i, which is the top variable of the inputs. For granularity G, a variable x_i is in the cache block $i \bmod G$. Then each BDD suboperation only uses the cache once for each cache block, by comparing the cache block of the parent operation and of the current operation.

This is a deterministic method to use the operation cache only sometimes rather than always. In practice, we see that this technique improves the performance of BDD operations.

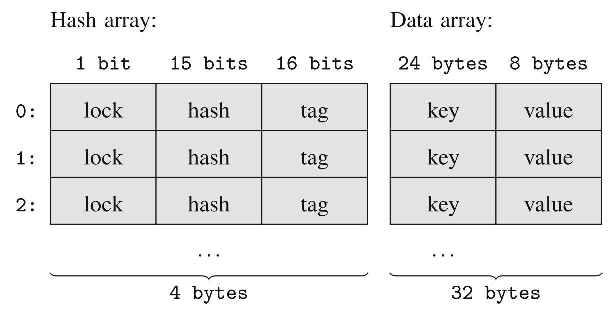


Fig. 10 Layout of the operation cache

If the granularity G is too large, the cost of recomputing results becomes too high, though, so care must be taken to keep G at a reasonable value.

We use an operation cache which, like the hash tables described above, consists of two arrays: the hash array and the data array. See Fig. 10 for the layout. Since we implement a lossy cache, the design of the operation cache is extremely simple. We do not implement a special strategy to deal with hash collisions, but simply overwrite the old results. There is a trade-off between the cost of recomputing operations and the cost of synchronizing with the cache. For example, the caching granularity increases the number of recomputed operations but improves the performance in practice.

The most important concern for correctness is that every result obtained via `cache-get` was inserted earlier with `cache-put`, and the most important concern for performance is that the number of memory accesses is as low as possible. To ensure this, we use a 16-bit "version tag" that increments (modulo 4096) with every update to the bucket, and check this value before reading and after reading the cache to check if the obtained result is valid. The chance of obtaining an incorrect result is astronomically small, as this requires precisely 4096 `cache-put` operations on the same bucket by other workers between the first and the second time the tag is read in `cache-get`, and the last of these 4096 other operations must have exactly the same hash value. Using a "version tag" like this is a well-known technique that goes back to as early as 1975 [36, p. 125].

We reserve 24 bytes of the bucket for the operation and its parameters. We use the first 64-bit value to store a BDD parameter and the operation identifier. The remaining 128 bits store other parameters, such as up to two 64-bit values, or up to three BDDs (123 bits, with 41 bits per BDD with a complement edge). The same holds for MTBDDs and LDDs. The result of the operation can be any 64-bit value or a BDD. Note that with 32 bytes per bucket and a properly aligned array, accessing a bucket requires only 1 cacheline transfer. As there are two buckets per cacheline, there is a tiny possibility for "false sharing" causing performance degradation, but due to the nature of hash tables, this should only rarely occur.

```
1 def cache-put(key, value):
2     h, location ← hash(key)
3     s ← harray[location]
4     if s.lock : return
5     if s.hash = h : return
6     if not cas(harray[location], s, {1, h, s.tag + 1}) : return
7     darray[location] ← {key, value}
8     harrray[location] ← {0, h, s.tag + 1}
```

Algorithm 4 The `cache-put` algorithm

See Algorithm 4 for the `cache-put` algorithm and Algorithm 5 for the `cache-get` algorithm. The algorithms are quite straight-forward. We use a 64-bit hash function that returns sufficient bits for the 15-bit `h` value and the `location` value. The `h` value is used for the hash in the hash array, and the `location` for the location of the bucket in the table. The `cache-put` operation aborts as soon as some problem arises, i.e., if the bucket is locked (line 4), or if the hash of the stored key matches the hash of the given key (line 5), or if the `cas` operation fails (line 6). If the `cas` operation succeeds, then the bucket is locked. The key-value pair is written to the cache array (line 7) and the bucket is unlocked (line 8, by setting the locked bit to 0).

In the `cache-get` operation, when the bucket is locked (line 4), we abort instead of waiting for the result. We also abort if the hashes are different (line 5). We read the result (line 6) and compare the key to the requested key (line 7). If the keys are identical, then we verify that the cache bucket has not been manipulated by a concurrent operation by comparing the "tag" counter (line 8).

As discussed above, it is possible that between lines 6–8 of the `cache-get` operation, exactly 4096 `cache-put` operations are performed on the same bucket by other workers, where the last one has exactly the same hash. The chances of this occurring are astronomically small. The reason we choose this design is that this implementation of `cache-get` only reads from memory and never writes. Memory writes cause additional communication between processors and with the memory when writing to the cacheline, and also force other processor caches to invalidate their copy of the bucket. We also want to avoid locking buckets for reading, because locking often causes bottlenecks. Since there are no loops in either algorithm, both algorithms are wait-free.

```
1 def cache-get(key):
2     h, location ← hash(key)
3     s ← harray[location]
4     if s.lock : return ⊥
5     if s.hash ≠ h : return ⊥
6     storedkey, value ← darray[location]
7     if storedkey ≠ key : return ⊥
8     if s ≠ harray[location] : return ⊥
9     return value
```

Algorithm 5 The `cache-get` algorithm

5.4 Garbage collection

Operations on decision diagrams typically create many new nodes and discard old nodes. Nodes that are no longer referenced are called "dead nodes". Garbage collection, which removes dead nodes from the unique table, is essential for the implementation of decision diagrams. Since dead nodes are often reused in later operations, garbage collection should be delayed as long as possible [55].

There are various approaches to garbage collection. For example, a *reference count* could be added to each node, which records how often a node is referenced. Nodes with a reference count of zero are either immediately removed when the count decreases to zero, or during a separate garbage collection phase. Another approach is *mark-and-sweep*, which marks all nodes that must be kept and removes all unmarked nodes. We refer to [55] for a more in-depth discussion of garbage collection.

For a parallel implementation, reference counts can incur a significant cost, as accessing nodes implies continuously updating the reference count, increasing the amount of communication between processors, as writing to a location in memory requires all other processors to refresh their view on that location. This is not a severe issue with only one processor, but with many processors this results in excessive communication, especially for nodes that are often used.

When parallelizing decision diagram operations, we can choose to perform garbage collection "on-the-fly", allowing other workers to continue inserting nodes, or we can "stop-the-world" and have all workers cooperate on garbage collection. We use a separate garbage collection phase, during which no new nodes are inserted. This greatly simplifies the design of the hash table, and we see no major advantage to allow some workers to continue inserting nodes during garbage collection.

Some decision diagram implementations use a global variable that counts how many buckets in the nodes table are in use and triggers garbage collection when a certain percentage of the table is in use. We want to avoid global counters like this and instead use a bounded probe sequence for the nodes table: when the algorithm cannot find an empty bucket in the first K buckets, garbage collection is triggered. In simulations and experiments, we find that this occurs when the hash table is between 80 and 95 % full.

As described above, decision diagram nodes are stored in a "data array", separated from the metadata of the unique table, which is stored in the "hash array". Nodes can be removed from the hash table without deleting them from the data array, simply by clearing the hash array. The nodes can then be reinserted during garbage collection, without changing their location in the data array, thus preserving the identity of the nodes.

We use a mark-and-sweep approach, where we keep track of all nodes that must be kept during garbage collection. Our implementation of parallel garbage collection consists of the following steps:

1. Initiate the operation using the Lace framework to arrange the "stop-the-world" interruption of all ongoing tasks.
2. Clear the hash array of the unique table, and clear the operation cache. The operation cache is cleared instead of checking each entry individually after garbage collection, although that is also possible.
3. Mark all nodes that we want to keep, using various mechanisms that keep track of the decision diagram nodes that we want to keep (see below).
4. Count the number of kept nodes and optionally increase the size of the unique table. Also optionally change the size of the operation cache.
5. Rehash marked nodes in the hash array of the unique table.

To mark all used nodes, Sylvan has a framework that allows custom mechanisms for keeping track of used nodes. During the "marking" step of garbage collection, the marking callback of each mechanism is called and all used decision diagram nodes are recursively marked. Sylvan itself implements four such mechanisms (also for MTBDDs and LDDs):

– The `sylvan_protect` and `sylvan_unprotect` methods maintain a set of pointers. During garbage collection, each pointer is inspected and the BDD is marked. This method is preferred for long-lived external references.
– Each thread has a thread-local BDD stack, operated using the methods `bdd_refs_push` and `bdd_refs_pop`. This method is preferred to store intermediate results in BDD operations.
– Each thread has a thread-local Task stack, operated using the methods `bdd_refs_spawn` and `bdd_refs_sync`. Tasks that return BDDs are stored in the stack, and during garbage collection the results of finished tasks are marked. This method is required when using `SPAWN` and `SYNC` on a task that returns a BDD.
– The `sylvan_ref` and `sylvan_deref` methods maintain a set of BDDs to be marked during garbage collection. This is a standard method offered by many BDD implementations, but we recommend using `sylvan_protect` and `sylvan_unprotect` instead.

To initiate garbage collection, we use a feature in the Lace framework that suspends all current work and starts a new task tree. This task suspension is a cooperative mechanism. Workers often check whether the current task tree is being suspended, either explicitly using the parallel framework, or implicitly when creating or synchronizing on tasks. Implementations of BDD operations make sure that all used BDDs are accounted for, typically with `bdd_refs_push` and `bdd_refs_spawn`, before such checks.

The garbage collection process itself is also executed in parallel. Removing all nodes from the hash table and clearing the operation cache is an instant operation that is amortized over time by the operating system by reallocating the memory (see below). Marking nodes that must be kept occurs in parallel, mainly by implementing the marking operation as a recursive task using Lace. Counting the number of used nodes and rehashing all nodes (steps 4–5) is also parallelized using a standard binary divide-and-conquer approach, which distributes the memory pages over all workers.

5.5 Memory management

Memory in modern computers is divided into regions called pages that are typically (but not always) 4096 bytes in size. Furthermore, computers have a distinction between "virtual" memory and "real" memory. It is possible to allocate much more virtual memory than we really use. The operating system is responsible for assigning real pages to virtual pages and clearing memory pages (to zero) when they are first used.

We use this feature to implement resizing of our unique table and operation cache. We preallocate memory according to a maximum number of buckets. Via global variables `table_size` and `max_size` we control which part of the allocated memory is actually used. When the table is resized, we simply change the value of `table_size`. To free pages, the kernel can be advised to free real pages using a `madvise` call (in Linux), but Sylvan only implements increasing the size of the tables, not decreasing their size.

Furthermore, when performing garbage collection, we clear the operation cache and the hash array of the unique table by reallocating the memory. Then, the actual clearing of the used pages only occurs *on demand* by the operating system, when new information is written to the tables.

6 Algorithms on decision diagrams

The current section discusses various operations that we implement in Sylvan on binary decision diagrams, multi-terminal binary decision diagrams and list decision diagrams.

6.1 BDD algorithms

Sylvan implements the basic BDD operations (Table 1) `and`, `not` and `xor`, the if-then-else (`ite`) operation, and

Table 1 Basic BDD operations on the input BDDs x, y, z

Operation	Implementation
$x \wedge y$	and(x, y)
$x \vee y$	not(and(not(x), not(y)))
$\neg(x \wedge y)$	not(and(x, y))
$\neg(x \vee y)$	and(not(x), not(y))
$x \oplus y$	xor(x, y)
$x \leftrightarrow y$	not(xor(x, y))
$x \rightarrow y$	not(and(x, not(y)))
$x \leftarrow y$	not(and(not(x), y))
if x then y else z	ite(x, y, z)
$\exists v$: x	exists(x, v)
$\forall v$: x	not(exists(not(x), v))

```
1  def and(x, y):
2      if $x = 1$ : return $y$
3      if $y = 1 \vee x = y$ : return $x$
4      if $x = 0 \vee y = 0 \vee x = \neg y$ : return 0
5      if result ← cache[$(x, y)$] : return result
6      $v$ = topvar($x$,$y$)
7      do in parallel:
8          low ← and($x_{v=0}$, $y_{v=0}$)
9          high ← and($x_{v=1}$, $y_{v=1}$)
10     result ← lookupBDDnode($v$, low, high)
11     cache[$(x, y)$] ← result
12     return result
```

Algorithm 6 Parallelized BDD algorithm and, with as parameters the BDDs x and y. The result is a BDD representing $x \wedge y$

```
1  def compose($x$, $M$):
2      if $x = 0 \vee x = 1 \vee M = 0$ : return $x$
3      $v$ = var($x$)
4      while $M \neq 0 \wedge$ var($M$) $< v$ : $M$ ← low($M$)
5      if $M = 0$ : return $x$
6      if result ← cache[$(x, M)$] : return result
7      do in parallel:
8          low ← compose(low($x$), $M$)
9          high ← compose(high($x$), $M$)
10     if $v$ = var(M) : result ← ite(high($M$), high, low)
11     else: result ← lookupBDDnode($v$, low, high)
12     cache[$(x, M)$] ← result
13     return result
```

Algorithm 7 Apply functional composition $x[M]$, where M is a mapping from variables to Boolean functions

exists. Implementing the basic operations in this way is common for BDD packages. Negation $\neg$ (not) is performed using complement edges, and is essentially free.

The parallelization of these functions is straightforward. See Algorithm 6 for the parallel implementation of and. This algorithm checks the trivial cases (lines 2–4) before the operation cache (line 5), and then runs the two independent suboperations (lines 8–9) in parallel.

Another operation that is parallelized similarly is the compose operation, which performs functional composition, i.e., substitute occurrences of variables in a Boolean formula by Boolean functions. For example, the substitution $[x_1 := x_2 \vee x_3, x_2 := x_4 \vee x_5]$ applied to the function $x_1 \wedge x_2$ results in the function $(x_2 \vee x_3) \wedge (x_4 \vee x_5)$. Sylvan offers a functional composition algorithm based on a "BDDMap". This structure is not a BDD itself, but uses BDD nodes to encode a mapping from variables to BDDs. A BDDMap is based on a disjunction of variables, but with the "high" edges going to BDDs instead of the terminal 1. This method also implements substitution of variables, e.g. $[x_1 := x_2, x_2 := x_3]$. See Algorithm 7 for the algorithm compose. This parallel algorithm is similar to the algorithms described above, with the composition functionality at lines 10–11. If the variable is in the mapping M, then we use the if-then-else method to compute the substitution. If the variable is not in the mapping M, then we simply compute the result using lookupBDDnode.

Sylvan also implements parallelized versions of the BDD minimization algorithms restrict and constrain (also called generalized cofactor), based on sibling-substitution, which are described in [20] and parallelized similarly as the and algorithm above.

Relational products In model checking using decision diagrams, relational products play a central role. Relational products compute the successors or the predecessors of (sets of) states. Typically, states are encoded using Boolean variables $\vec{x} = x_1, x_2, \ldots, x_N$. Transitions between these states are represented using Boolean variables $\vec{x}$ for the source states and variables $\vec{x}' = x'_1, x'_2, \ldots, x'_N$ for the target states. Given a set of states S_i encoded as a BDD on variables $\vec{x}$, and a transition relation R encoded as a BDD on variables $\vec{x} \cup \vec{x}'$, the set of states S'_{i+1} encoded on variables $\vec{x}'$ is obtained by computing $S'_{i+1} = \exists \vec{x}\colon (S_i \wedge R)$. BDD packages typically implement an operation and_exists that combines $\exists$ and $\wedge$ to compute S'_{i+1}.

Typically, we want the BDD of the successors states defined on the unprimed variables $\vec{x}$ instead of the primed variables $\vec{x}'$, so the and_exists call is then followed by a variable substitution that replaces all occurrences of variables from $\vec{x}'$ by the corresponding variables from $\vec{x}$. Furthermore, the variables are typically interleaved in the variable ordering, like $x_1, x'_1, x_2, x'_2, \ldots, x_N, x'_N$, as this often results in smaller BDDs. Sylvan implements specialized operations relnext and relprev that compute the successors and the predecessors of sets of states, where the transition relation is encoded with the interleaved variable ordering. See Algorithm 8 for the implementation of relnext. This function takes as input a set S, a transition relation R, and the set of variables V, which is the union of the interleaved sets $\vec{x}$ and $\vec{x}'$ (the variables on which the transition relation is defined). We

```
1  def relnext($S$, $R$, $V$):
2      if $S = 0 \lor R = 0$ : return 0
3      if $S = 1 \land R = 1$ : return 1
4      $v$ = topvar($S$,$R$)
5      while var($V$) < $v$ : $V \leftarrow$ next($V$)
       // if $V = \emptyset$, we assume $R$ is irrelevant
6      if $V = \emptyset$ : return $S$
7      if result $\leftarrow$ cache[($S$, $R$, $V$)] : return result
8      if $v$ = var($V$) :
9          x, x' $\leftarrow$ unprimed $v$, primed $v$
10         V' $\leftarrow$ V without x and x'
11         do in parallel:
12             a $\leftarrow$ relnext($S_{x=0}$, $R_{x=0,x'=0}$, $V'$)
13             b $\leftarrow$ relnext($S_{x=1}$, $R_{x=1,x'=0}$, $V'$)
14             c $\leftarrow$ relnext($S_{x=0}$, $R_{x=0,x'=1}$, $V'$)
15             d $\leftarrow$ relnext($S_{x=1}$, $R_{x=1,x'=1}$, $V'$)
16         do in parallel:
17             low $\leftarrow$ or(a, b)
18             high $\leftarrow$ or(c, d)
19         result $\leftarrow$ lookupBDDnode($x$, low, high)
20     else:
           // $v$ is not in $R$, by assumption
21         do in parallel:
22             low $\leftarrow$ relnext($S_{v=0}$, $R$, $V$)
23             high $\leftarrow$ relnext($S_{v=1}$, $R$, $V$)
24         result $\leftarrow$ lookupBDDnode($v$, low, high)
25     cache[($S$, $R$, $V$)] $\leftarrow$ result
26     return result
```

Algorithm 8 The parallel algorithm `relnext`, which given the BDDs S (representing a set of states), R (representing a transition relation) and V (the cube of interleaved variables $\vec{x} \cup \vec{x}'$) computes the set of successor states defined on $\vec{x}$, i.e., $\big(\exists \vec{x} : (S \land R)\big)[\vec{x}' := \vec{x}]$. We assume that all variables in R are also in V

first check for terminal cases (lines 2–3). These are the same cases as for the $\land$ operation. Then we process the set of variables V to skip variables that are not in S and R (lines 5–6). After consulting the cache (line 7), either the current variable is in the transition relation, or it is not. If it is not, we perform the usual recursive calls and compute the result (lines 21–24). If the current variable is in the transition relation, then we let x and x' be the two relevant variables (either of these equals v) and compute four subresults, namely for the transition (a) from 0 to 0, (b) from 1 to 0, (c) from 0 to 1, and (d) from 1 to 1 in parallel (lines 11–15). We then abstract from x' by computing the existential quantifications in parallel (lines 16–18), and finally compute the result (line 19). This result is stored in the cache (line 25) and returned (line 26). We implement `relprev` similarly.

6.2 MTBDD algorithms

Although multi-terminal binary decision diagrams are often used to represent functions to integers or real numbers, they could be used to represent functions to any domain. In practice, the well-known BDD package CUDD [56] implements MTBDDs with `double` (floating-point) leaves. For some applications, other types of leaves are required, for example to represent rational numbers or integers. To allow different types of MTBDDs, we designed a generic customizable framework. The idea is that anyone can use the given functionality or extend it with other leaf types or other operations.

By default, Sylvan implements five types of leaves:

Leaf type	Function type
BDDs `false` and `true`	Total functions $\mathbb{B}^N \to \mathbb{B}$ (BDDs)
64-bit integer (`uint64`)	Partial functions $\mathbb{B}^N \to \mathbb{N}$
Floating-point (`double`)	Partial functions $\mathbb{B}^N \to \mathbb{R}$
Rational leaves	Partial functions $\mathbb{B}^N \to \mathbb{Q}$
GMP library leaves (`mpq`)	Partial functions $\mathbb{B}^N \to \mathbb{Q}$

The BDDs `false` and `true` (complemented `false`) are not encoded as MTBDD leaves as in Sect. 5.1, but we reuse the BDD 0 that is reserved for the leaf `false`. For the rational leaves we use 32 bits for the numerator and 32 bits for the denominator. Sylvan also implements the leaf type `mpq` which uses the GMP library for arbitrary precision arithmetic, i.e., an arbitrary number of bits for the numerator and the denominator. The framework supports partially defined functions, reusing the BDD `false` to mean $\perp$ for non-Boolean functions.

Sylvan implements a generic binary apply function, a generic monadic apply function, and a generic abstraction algorithm. The implementation of binary apply is similar to Algorithm 1. See Algorithm 9 for the implementation of abstraction. On top of these generic algorithms, we implemented basic operators `plus`, `times`, `min` and `max` for the default leaf types. For all valuations of MTBDDs x and y that end in leaves a and b, they compute $a + b$, $a \times b$, $\min(a, b)$ and $\max(a, b)$. For Boolean MTBDDs, the `plus` and `times` operators are similar to $\lor$ and $\land$. When using

```
1  def abstract($x$, $V$, F):
2      if $x = 0 \lor x = 1 \lor V = \emptyset$ : return $x$
3      if result $\leftarrow$ cache[($x$, $V$, F)] : return result
4      if $x$ is a leaf or var($V$) < topvar($x$) :
5          sub $\leftarrow$ abstract($x$, next($V$), $F$)
6          result $\leftarrow$ F(sub, sub)
7      elif var($V$) = topvar($x$) :
8          do in parallel:
9              low $\leftarrow$ abstract($x_{v=0}$, next($V$), F)
10             high $\leftarrow$ abstract($x_{v=1}$, next($V$), F)
11         result $\leftarrow$ F(low, high)
12     else:
13         do in parallel:
14             low $\leftarrow$ abstract($x_{v=0}$, $V$, F)
15             high $\leftarrow$ abstract($x_{v=1}$, $V$, F)
16         result $\leftarrow$ lookupMTBDDnode($v$, low, high)
17     cache[($x$, $V$, F)] $\leftarrow$ result
18     return result
```

Algorithm 9 Parallel MTBDD algorithm that applies the abstraction F for the variables in V

`times` with a Boolean MTBDD (or a BDD) and an MTBDD of some other type, it acts as a filter, removing the subgraphs where the BDD is `false`.

Sylvan supports custom leaves with 64-bit values. These 64-bit values can also be pointers. In that case, for the canonical representation of leaves it is not sufficient to compare the 64-bit values, which is the default behavior. Also, the pointers typically point to dynamically allocated memory that must be freed when the leaf is deleted. To support custom leaves, Sylvan implements a framework where custom callbacks are registered for each leaf type. These custom callbacks are:

- `hash(value, seed)` computes a 64-bit hash for the leaf value and the given 64-bit seed.
- `equals(value1, value2)` returns 1 if the two values encode the same leaf, and 0 otherwise. The default implementation simply compares the two values.
- `create(pointer)` is called when a new leaf is created with the 64-bit value references by the pointer; this allows implementations to allocate memory and replace the referenced value with the final value.
- `destroy(value)` is called when the leaf is garbage collected so the implementation can free memory allocated by `create`.

We use this functionality to implement the GMP leaf type. The GMP leaf type is essentially a pointer to a different datastructure to support arbitrary precision arithmetic. The above functions are implemented as follows:

- `hash` follows the pointer and hashes the contents of the `mpq` datastructure.
- `equals` follows the pointers and compares their contents.
- `create` clones the `mpq` datastructure and writes the address of the clone to the given new leaf.
- `destroy` frees the memory of the cloned datastructure.

6.3 LDD algorithms

We implemented various LDD operations that are required for model checking in LTSmin (see Sect. 7), such as the set operations `union`, `intersect`, and `minus`. We implemented `project` (existential quantification), `enumerate` (for enumeration of elements in a set) and the two relational operations `relprod` and `relprev`. 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.

7 Application: parallelism in LTSmin

One major application for which we developed Sylvan is symbolic model checking. This section describes one of the main algorithms in symbolic model checking, which is symbolic reachability. We describe the implementation of on-the-fly symbolic reachability in the model checking toolset LTSmin, and show how we parallelize symbolic reachability using the disjunctive partitioning of transition relations that LTSmin offers, and how we parallelize on-the-fly transition learning using a custom BDD operation.

7.1 On-the-fly symbolic reachability in LTSmin

In model checking, we create models 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. Many model checking algorithms depend on state-space generation using a reachability algorithm, for example to calculate all states that are reachable from the initial state of the system, or to check if an invariant is always true, and so forth.

The Pins *interface* The model checking toolset LTSmin provides a language independent Partitioned Next-State Interface (Pins), which connects various input languages to model checking algorithms [7,24,38,42,44]. In Pins, the states of a system are represented by vectors of N integer values. Furthermore, transitions are distinguished in K disjunctive "transition groups", i.e., each transition in the system belongs to one of these transition groups. The transition relation of each transition group usually only depends on a subset of the entire state vector called the "short vector". This enables the efficient encoding of transitions that only affect some integers of the state vector. Variables in the short vector are further distinguished by the notions of read dependency and write dependency [44]: the variables that are inspected or read to obtain new transitions are in the "read vector" of the transition group, and the variables that can be modified by transitions in the transition group are in the "write vector". An example of a variable that is only in the read vector is a guard; when a variable is only in the write vector, then its original value is irrelevant. Computing short vectors from long vectors is called "projection" in LTSmin and is similar to existential quantification.

Learning transitions Initially, LTSmin does not have knowledge of the transitions in each transition group, and only the initial state is known. As the model is explored, new transitions are learned via the Pins interface and added to the transition relation. Every Pins language module implements a `next-state` function. This `next-state` function takes as input the source state (as a read vector) and the transition group. The `next-state` function then produces all target states (as write vectors) that can be reached from the source state. Algorithms in LTSmin thus learn new transitions on-the-fly. Internally, LTSmin offers various backends

```
1  def reachable(initial):
       // global variables: relations[K], K
2      states ← initial
3      frontier ← initial
4      while frontier ≠ ∅ :
5          for k ∈ {0, ..., K−1} :
6              learn-transitions (frontier, k)
7              next[k] ← relnext (frontier, relations[k])
8          frontier ← big-union (next, 0, K)
9          frontier ← minus (frontier, states)
10         states ← union (states, frontier)
11     return states
```

Algorithm 10 Symbolic on-the-fly reachability algorithm (using a frontier set) with K transition groups. Computes the set of states reachable from the initial state. The transition relations are updated with on-the-fly learning (line 6)

to store discovered states and transitions, including binary decision diagrams and list decision diagrams from Sylvan. With list decision diagrams, integers from the state vector can be used directly in the decision diagram nodes. With binary decision diagrams, the integers must be represented in binary form, typically using a fixed number of bits per integer.

The symbolic reachability algorithm with K transition groups and on-the-fly learning is given in Algorithm 10. This algorithm is an extension of the standard breadth-first-search (BFS) reachability algorithm with a frontier set. Algorithm 10 iteratively discovers new states until no new states are found (line 4). For every transition group (line 5), the transition group is updated with new transitions learned from the frontier set (line 6). The updated transition relation `relations[k]` is then used to symbolically find all successors of the states in the frontier set (line 7). This uses the `relnext` operation that is described in Sect. 6.1. The sets of new states discovered for every transition group are pair-wise merged into the new set `frontier` (line 8). Successors that have been found in earlier iterations are removed (line 9). All new states are then added to the set of discovered states `states` (line 10). When no new states are discovered, the set of discovered states is returned (line 11).

7.2 Parallel on-the-fly symbolic reachability

Even with parallel BDD operations, the parallel speedup of model checking in LTSMIN is limited, especially for smaller models, where the size of "work units" (between sequential points in the algorithm) is small and when there are few independent tasks. Experiments in [23] demonstrate this limitation. This is expected: if a parallel program consists of many small operations between sequential points, or if small input BDDs result in few independent tasks, then we expect limited parallel scalability.

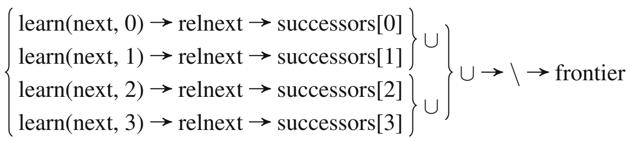

learn(next, 0) → relnext → successors[0]
learn(next, 1) → relnext → successors[1]
learn(next, 2) → relnext → successors[2]
learn(next, 3) → relnext → successors[3]
∪
∪
∪ → \ → frontier

Fig. 11 Schematic overview of parallel on-the-fly reachability

```
1  def par-next (frontier, i, k):
2      if k = 1 :
3          learn-transitions (frontier, i)
4          next ← relnext (frontier, relations[i])
5          return next
6      else:
7          do in parallel:
8              left ← par-next (frontier, i, k/2)
9              right ← par-next (frontier, i + k/2, k − k/2)
10         return union (left, right)
11 def reachable (initial):
12     states ← initial
13     frontier ← initial
14     while frontier ≠ ∅ :
15         frontier ← par-next (frontier, 0, K)
16         frontier ← minus (frontier, states)
17         states ← union (states, frontier)
18     return states
```

Algorithm 11 Parallel symbolic on-the-fly reachability with K transition groups

Since LTSMIN partitions the transition relation in transition groups, many small BDD operations are executed in sequence, for each transition group. To improve the parallel speedups, we execute lines 5–9 of Algorithm 10 in parallel, as in Fig. 11. See Algorithm 11 for the resulting algorithm for parallel reachability. 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.

In addition, we parallelize the learning algorithm, using a special BDD algorithm `collect` that combines enumeration and union. In the new implementation, the callback for enumeration does not add the learned transitions to the transition relation, which would result in race conditions, but returns the learned transitions as a BDD or LDD. These sets of transitions are then merged by `collect`. See Algorithm 12. This algorithm uses the "states" BDD and the set of variables "vars" to generate all state vectors "vec". For every state in the set of states, a callback is called (line 2). The callback `next-state` returns a BDD containing the transitions from the given short state. All learned transitions are then merged and returned (line 8).

Using the framework offered by Sylvan, parallel on-the-fly symbolic reachability is straightforward to implement. We use the Lace work-stealing framework to parallelize certain parts of the reachability algorithm. The `collect` method is the only custom decision diagram operation needed and

```
def collect(states, vars, callback, vec={}):
    if vars = ∅ : return callback(vec)
    if states = false : return ∅
    v, vars ← head(vars), tail(vars)
    do in parallel:
        R0 ← collect(states_{v=0}, vars, callback, vec+{0})
        R1 ← collect(states_{v=1}, vars, callback, vec+{1})
    return union(R0, R1)
```

Algorithm 12 The parallel `collect` algorithm (BDD version) combining `enumerate` and `union`. The callback is called for every state and returns the set of transitions from that state to its successors. The returned sets are pairwise merged and returned

is about 25 lines of code long, including some overhead to manage internal references for garbage collection.

8 Experimental evaluation

In the current section, we evaluate the LDD extension of Sylvan, and the application of parallelization to LTSMIN. Compared to [26], we add benchmark results with the new unique table, using LDDs and the fully parallel strategy. We use the same machine as in [26] and confirmed that the original benchmarks still yield comparable results.

The experimental evaluation is based on the BEEM model database [51]. Of the 300 models, 269 were successfully explored in [26]. The `plc` and `train-gate` models had an unfortunate parsing error in the PINS wrapper. Several other models timed out (with a timeout of 1200 s). We perform the experiments on a 48-core machine, consisting of 4 AMD Opteron™ 6168 processors with 12 cores each and 128 GB of internal memory. We perform symbolic reachability in LTSMIN toolset with the following command:

```
dve2lts-sym -rgs --order=<order>
                 --vset=lddmc <model>.dve
```

We select as the fixed size of the unique table 2^{30} buckets and as of the operation cache also 2^{30} buckets (24 GB for the unique table and 36 GB for the operation cache). Using the parameter `-order` we either select the `par-prev` variation or the `bfs-prev` variation. The `bfs-prev` variation does not have parallelism in LTSMIN, but 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 computing the successors are performed for all transition groups in parallel.

The full experiment data and benchmark scripts can be found online.[1]

[1] For the benchmarks from [26] at https://github.com/utwente-fmt/sylvan-tacas2015 and at https://github.com/utwente-fmt/sylvan-sttt for the benchmarks presented here.

8.1 Comparing **`par-prev`** and **`bfs-prev`**

The following table summarizes the runtimes on all 269 benchmark models using the `bfs-prev` and `par-prev` variations, and shows in particular the results of a few selected instances. We measure the time spent to execute symbolic reachability, excluding time spent initializing LTSMIN. Each data point is the average of at least three measurements. We used the old version of the hash table [26] to obtain these results.

Experiment	T_1	T_{48}	T_1/T_{48}
`blocks.4` (par)	629.54	16.58	38.0
`blocks.4` (bfs)	630.04	21.69	29.0
`lifts.8` (par)	377.52	12.03	31.4
`lifts.8` (bfs)	377.36	26.11	14.5
`firewire_tree.1` (par)	16.40	0.99	16.5
`firewire_tree.1` (bfs)	16.43	11.35	1.4
Sum of all `par-prev`	20,756	1298	16.0
Sum of all `bfs-prev`	20,745	3737	5.6

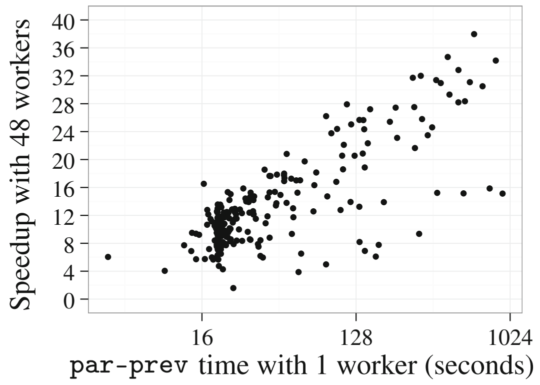
Model `blocks.4` results in the highest speedup of 38.0×. The model `lifts.8` has a speedup of 14.5× with `bfs-prev` and more than twice with `par-prev`. The overhead (difference in T_1) between the "sequential" `bfs-prev` and "parallel" `par-prev` versions is negligible. For all models, the speedup either improves with `par-prev`, or stays the same.

For an overview of the obtained speedups on the entire benchmark set, see Fig. 12. Here we see that "larger" models (higher T_1) are associated with a higher parallel speedup. This plot also shows the benefit of adding parallelism on the algorithmic level, as many models in the fully parallel version have higher speedups. One of the largest improvements was obtained with the `firewire_tree.1` model, which went from 1.4× to 16.5×. We conclude that the lack of parallelism is a bottleneck, which can be alleviated by exploiting the disjunctive partitioning of the transition relation.

8.2 Comparing the old and the new unique table

We repeated the benchmarks on all 269 benchmark models using the `par-prev` variation and the new unique table. Each new data point is the average of at least 18 measurements.

Experiment	T_1	T_{48}	T_1/T_{48}
Sum of all benchmarks (old)	20,756	1298	16.0
Sum of all benchmarks (now)	16,357	907	18.0



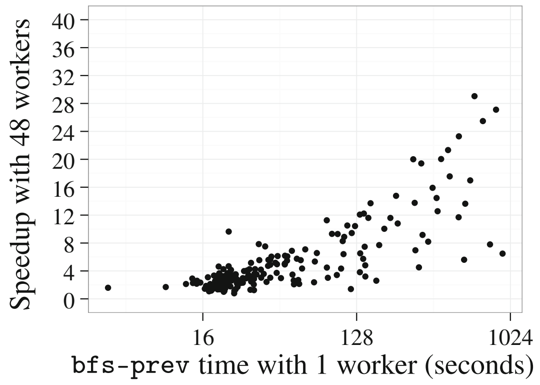


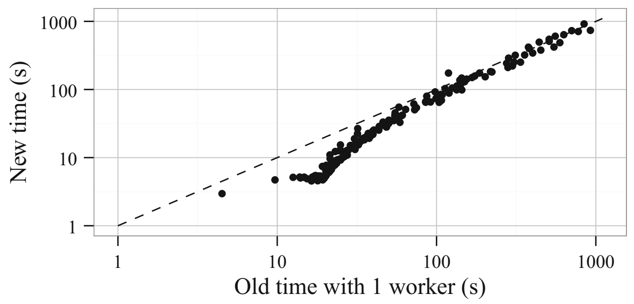
Fig. 12 Results of the 269 benchmark models, with fully parallel learning and parallel transition groups (*above*), and with only parallel BDD operations (*below*)

We observe that the benchmarks run faster and result in an improved parallel speedup. We suggest that this is mostly due to the different unique table design, as there are no major changes to the implementation of LDD operations or in LTSMIN for these benchmarks.

For a more insightful comparison of the previous and the current benchmark results, see Fig. 13. The results suggest that smaller benchmark models benefit more from the new hash table design. See Fig. 14 for a speedup graph of a selection of the models with the highest speedups. The point of this speedup graph is that most likely further speedups would be obtained after 48 cores for the selected models. In earlier work [23], we already determined that the obtained speedup strongly depends on the size of the models.

8.3 Comparing BDDs and LDDs

Finally, we compared the performance of our multi-core BDD and LDD variants for the `par-prev` variation of on-the-fly symbolic reachability. Figure 15 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 $T_1 = 61.01$, $T_{48} = 9.36$. The large difference suggests that LDDs are a more efficient representation for the models of the BEEM database.



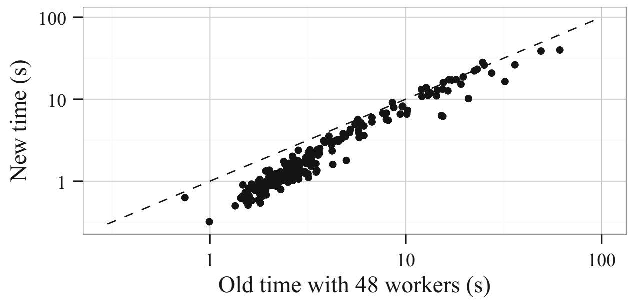


Fig. 13 Comparison between the benchmark results of [26] and new results, for 1 worker (*above*) and for 48 workers (*below*)

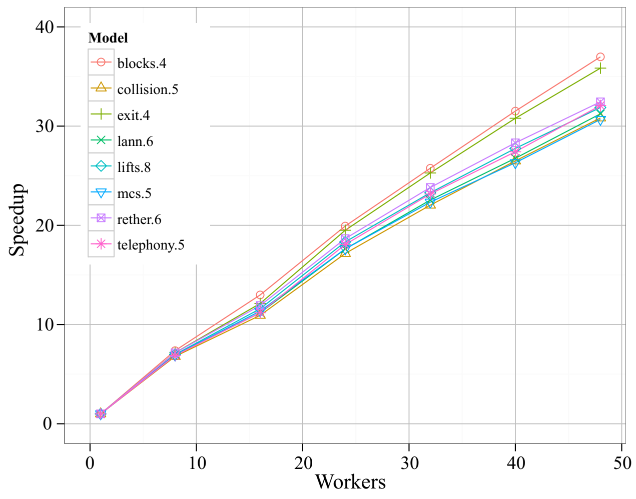


Fig. 14 Speedup graphs of some of the models, using LDDs and the fully parallel strategy. Each data point is an average of at least 18 measurements

8.4 Recent experiments in related work

Sylvan has also been used as a symbolic backend in the model checker ISCASMC, a probabilistic model checker [34] written in Java. A recent study [22] compared the performance of the BDD libraries CUDD, BuDDy, CacBDD, JDD, Sylvan, and BeeDeeDee when used as the symbolic backend of ISCASMC

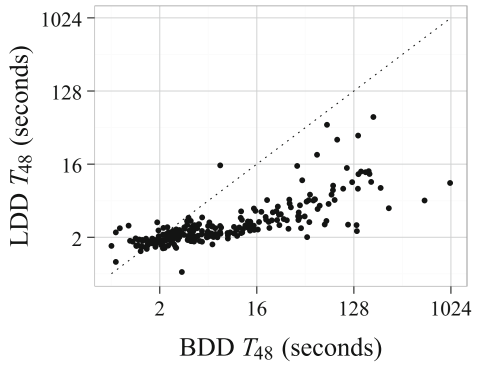


Fig. 15 Results of the models that did not time out for both BDDs and LDDs, comparing time with 48 workers

and performing symbolic reachability. They summarize the overall runtimes on all experiments (excluding failures) by the following table [22]:

Backend	Time (s)	Backend	Time (s)
sylvan-7	608	buddy	2156
cacbdd	1433	jdd	2439
cudd-bdd	1522	beedeedee	2598
sylvan-1	1838	cudd-mtbdd	2837

This result was produced with a version of Sylvan before the extensions that we present in the current paper. As the results show, Sylvan is competitive with other BDD implementations when used sequentially (with 1 worker) and benefits from parallelism (with 7 workers they obtained a speedup of 3×).

Recently, we used Sylvan for the implementation of symbolic bisimulation minimization [27]. For this particular application of binary decision diagrams, it is very beneficial to develop custom BDD operations and to use the MTBDD implementation we present in the current paper, especially for Continuous Time Markov Chains and Interactive Markov Chains models, for which support for rational numbers in the MTBDD leaves is highly preferred. Compared to the state of the art tool SIGREF [59] that relies on a version of CUDD, we obtained a sequential speedup of up to 95× and a parallel speedup of up to 17× using 48 workers on benchmarks from the literature [27].

9 Conclusion

This paper presented the design and implementation of a multi-core framework for decision diagrams, called Sylvan. Sylvan already supported parallel operations for (binary) BDDs and (multi-way) MDDs in the form of (list) LDDs. The most recent extension is the support for multiple terminals, i.e., a new framework for MTBDDs that supports various types of MTBDD leaves and was designed for customization. We discussed several BDD and MTBDD operations and offered an example implementation of custom MTBDD leaves and operations using the GMP library. We also discussed a new hash table design for Sylvan and showed clear improvements over the previous version. The new table supports parallelized garbage collection for decision diagrams and offers extensive support for customizable "marking" mechanisms.

Using Sylvan, one can very easily speedup sequential symbolic algorithms, by replacing the BDD operations by calls to their parallel implementation in Sylvan. On top of this, the framework also supports the parallelization of the higher-level algorithm itself, by allowing concurrent calls to BDD operations. This integration is based on our customizable work-stealing task scheduler Lace. We demonstrated this for parallel symbolic model checking with LTSMIN. Experimentally, we demonstrated a speedup of up to 38× (with 48 cores) for fully parallel on-the-fly symbolic reachability in LTSMIN, and an average of 18× for all the BEEM benchmark models using the new hash table.

Initially, it was not clear whether a BDD package could significantly profit from parallelization on a multi-core shared memory computer. BDD operations are highly memory-intensive and show irregular memory access patterns, so they belong to the hardest class to achieve practical speedup. We believe that there are three ingredients that enabled us to achieve efficient parallelism: The first is that we adapted the scalable, lockless hash-table design that already proved its value in explicit-state model checking [41]. The second is that we carefully designed the work-stealing task scheduler Lace [25] to handle the very small individual BDD steps. Finally, we followed a pragmatic approach: for instance, we just give up cache operations in case of race conditions, rather than retrying them. The experiments in this paper show that further improvements in the concurrent hash-table and cache design not only led to reduced running times sequentially, but even to a higher speedup.

Our measurements in recent papers and in the current paper show that sequential symbolic algorithms benefit from the "automatic" parallelization provided by the parallel decision diagram operations in Sylvan. We also demonstrated that adding parallelism to higher-level algorithms can result in even higher speedups. In general, multi-core decision diagrams can speed up symbolic model checking considerably.

Sylvan is available online[2] and is released under the Apache 2.0 License, so 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. A

[2] See https://github.com/utwente-fmt/sylvan.

more elaborate example of how applications can add custom BDD operations can be found in the SIGREFMC[3] tool.

An interesting extension of our work would be the addition of dynamic variable reordering. This could either be achieved by a parallel implementation of the sifting algorithm [52], or by a parallel investigation of different variable orderings. Another interesting research question would be to investigate whether the experimental speedups can be maintained for smarter exploration strategies. These strategies determine in which order the symbolic sub-transitions are fired. We have investigated the BFS and chaining strategies, since they are external to the decision diagram implementation. It is still open whether the full saturation strategy [16] also profits from parallelization, but this requires a tight integration of the exploration strategy into the multi-core decision diagram operations.

References

1. Acar, U.A., Charguéraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. In: PPOPP, pp. 219–228. ACM (2013)
2. Akers, S.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
3. Arunachalam, P., Chase, C.M., Moundanos, D.: Distributed binary decision diagrams for verification of large circuit. In: ICCD, pp. 365–370 (1996)
4. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: ICCAD 1993, pp. 188–191 (1993)
5. Bianchi, F., Corno, F., Rebaudengo, M., Reorda, M.S., Ansaloni, R.: Boolean function manipulation on a parallel system using BDDs. In: HPCN Europe, pp. 916–928 (1997)
6. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: ICTAC, LNCS, vol. 5160, pp. 81–95. Springer (2008)
7. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: CAV, LNCS, vol. 6174, pp. 354–359. Springer (2010)
8. Blumofe, R.D.: Scheduling multithreaded computations by work stealing. In: FOCS, pp. 356–368. IEEE Computer Society (1994)
9. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
10. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC, pp. 40–45 (1990)
11. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
12. Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
13. Cabodi, G., Gai, S., Sonza Reorda, M.: Boolean function manipulation on massively parallel computers. In: Proceedings of 4th Symposium on Frontiers of Massively Parallel Computation, pp. 508–509. IEEE (1992)
14. Chen, J., Banerjee, P.: Parallel construction algorithms for BDDs. In: ISCAS 1999, pp. 318–322. IEEE (1999)
15. Chung, M.Y., Ciardo, G.: Saturation NOW. In: QEST, pp. 272–281. IEEE Computer Society (2004)
16. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: an efficient iteration strategy for symbolic state-space generation. In: TACAS, LNCS, vol. 2031, pp. 328–342 (2001)
17. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: Saturation unbound. In: TACAS 2003, pp. 379–393 (2003)
18. Ciardo, G., Zhao, Y., Jin, X.: Parallel symbolic state-space exploration is difficult, but what is the alternative? In: PDMC, pp. 1–17 (2009)
19. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral transforms for large Boolean functions with applications to technology mapping. In: DAC, pp. 54–60 (1993)
20. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: ICCAD 1990, pp. 126–129. IEEE Computer Society (1990)
21. van Dijk, T.: Sylvan: Multi-core decision diagrams. Ph.D. thesis, University of Twente (2016)
22. van Dijk, T., Hahn, E.M., Jansen, D.N., Li, Y., Neele, T., Stoelinga, M., Turrini, A., Zhang, L.: A comparative study of BDD packages for probabilistic symbolic model checking. In: SETTA, LNCS, vol. 9409, pp. 35–51. Springer (2015)
23. van Dijk, T., Laarman, A., van de Pol, J.: Multi-core BDD operations for symbolic reachability. ENTCS **296**, 127–143 (2013)
24. van Dijk, T., Laarman, A.W., van de Pol, J.: Multi-core and/or symbolic model checking. ECEASST **53** (2012)
25. van Dijk, T., van de Pol, J.: Lace: non-blocking split deque for work-stealing. In: MuCoCoS, LNCS, vol. 8806, pp. 206–217. Springer (2014)
26. van Dijk, T., van de Pol, J.: Sylvan: Multi-core decision diagrams. In: TACAS, LNCS, vol. 9035, pp. 677–691. Springer (2015)
27. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. In: TACAS, LNCS, vol. 9636, pp. 332–348. Springer (2016)
28. Ezekiel, J., Lüttgen, G., Ciardo, G.: Parallelising symbolic state-space generators. In: CAV, LNCS, vol. 4590, pp. 268–280 (2007)
29. Faxén, K.: Efficient work stealing for fine grained parallelism. In: ICPP 2010, pp. 313–322. IEEE Computer Society (2010)
30. Faxén, K.F.: Wool—a work stealing library. SIGARCH Comput. Archit. News **36**(5), 93–100 (2008)
31. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI, pp. 212–223. ACM (1998)
32. Gai, S., Rebaudengo, M., Sonza Reorda, M.: An improved data parallel algorithm for Boolean function manipulation using BDDs. In: Proceedings of Euromicro Workshop on Parallel and Distributed Processing, pp. 33–39. IEEE (1995)
33. Grumberg, O., Heyman, T., Schuster, A.: A work-efficient distributed algorithm for reachability analysis. Form. Methods Syst. Des. **29**(2), 157–175 (2006)
34. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasmc: A web-based probabilistic model checker. In: FM, LNCS, vol. 8442, pp. 312–317. Springer (2014)
35. Heyman, T., Geist, D., Grumberg, O., Schuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: CAV, LNCS, vol. 1855, pp. 20–35. Springer, Berlin/Heidelberg (2000)

[3] See https://github.com/utwente-fmt/sigrefmc.

36. IBM: IBM System/370, Principles of Operation. IBM Publication No. GA22-7000-4 (1975)
37. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: theory and applications. Mult. Valued Log. **4**(1), 9–62 (1998)
38. 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)
39. Kimura, S., Clarke, E.M.: A parallel algorithm for constructing binary decision diagrams. In: Proceedings of International Conference on Computer Design: VLSI in Computers and Processors ICCD, pp. 220–223 (1990)
40. Kimura, S., Igaki, T., Haneda, H.: Parallel binary decision diagram manipulation. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E75-A**(10), 1255–1262 (1992)
41. Laarman, A., van de Pol, J., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: FMCAD 2010, pp. 247–255. IEEE (2010)
42. Laarman, A.W., van de Pol, J., Weber, M.: Multi-core LTSmin: marrying modularity and scalability. In: NFM 2011, LNCS, vol. 6617, pp. 506–511. Springer (2011)
43. Lovato, A., Macedonio, D., Spoto, F.: A thread-safe library for binary decision diagrams. In: SEFM, LNCS, vol. 8702, pp. 35–49. Springer (2014)
44. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC, LNCS, vol. 8855, pp. 204–219. Springer (2014)
45. Miller, D.M., Drechsler, R.: On the construction of multiple-valued decision diagrams. In: ISMVL, pp. 245–253 (2002)
46. Milvang-Jensen, K., Hu, A.J.: BDDNOW: a parallel BDD package. In: FMCAD, pp. 501–507 (1998)
47. Ochi, H., Ishiura, N., Yajima, S.: Breadth-first manipulation of SBDD of Boolean functions for vector processing. In: DAC, pp. 413–416 (1991)
48. Oortwijn, W.: Distributed symbolic reachability analysis. Master's thesis, University of Twente, Dept. of C.S. (2015)
49. Ossowski, J.: JINC—a multi-threaded library for higher-order weighted decision diagram manipulation. Ph.D. thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn (2010)
50. Parasuram, Y., Stabler, E.P., Chin, S.K.: Parallel implementation of BDD algorithms using a distributed shared memory. In: HICSS, vol. 1, pp. 16–25 (1994)
51. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN, pp. 263–267. Springer-Verlag, Berlin, Heidelberg (2007)
52. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: ICCAD, pp. 42–47 (1993)
53. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: High performance BDD package by exploiting memory hiercharchy. In: DAC, pp. 635–640 (1996)
54. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
55. Somenzi, F.: Efficient manipulation of decision diagrams. STTT **3**(2), 171–181 (2001)
56. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/ (2015)
57. Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: DAC, pp. 641–644 (1996)
58. Velev, M.N., Gao, P.: Efficient parallel GPU algorithms for BDD manipulation. In: ASP-DAC, pp. 750–755. IEEE (2014)
59. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref—a symbolic bisimulation tool box. In: ATVA, LNCS, vol. 4218, pp. 477–492. Springer (2006)
60. Yang, B., O'Hallaron, D.R.: Parallel breadth-first BDD construction. In: PPOPP, pp. 145–156 (1997)