Multi-core Decision Diagrams

Tom van Dijk and Jaco van de Pol

Abstract Decision diagrams are fundamental data structures that revolutionized fields such as model checking, automated reasoning and decision processes. As performance gains in the current era mostly come from parallel processing, an ongoing challenge is to develop data structures and algorithms for modern multi-core architectures. This chapter describes the parallelization of decision diagram operations as implemented in the parallel decision diagram package Sylvan, which allows sequential algorithms that use decision diagrams to exploit the power of multi-core machines.

1 Introduction

Decision diagrams are fundamental data structures in computer science and find applications in many areas. They are extensively used in symbolic model checking [15, 16], logic synthesis [40, 41, 55], Boolean satisfiability, fault tree analysis [52, 12], test generation [6, 1] and even to represent access control lists [26]. A recent survey paper by Minato [44] provides an accessible history of research into decision diagrams, listing applications to data mining [38], Bayesian networks and probabilistic inference models [45, 32], and game theory [53].

In the past, the processing power of computers increased mostly by improvements in the clock speed and the efficiency of processors, which often do not require adaptations to algorithms. However, as physical constraints seem to limit such improvements, further increases in processing power of modern machines inevitably

Tom van Dijk
Institute for Formal Methods and Verification, Johannes Kepler University, Linz, Austria
e-mail: tom.vandijk@jku.at

Jaco van de Pol
Formal Methods and Tools, University of Twente, Enschede, The Netherlands
e-mail: j.c.vandepol@utwente.nl

come from using multiple cores. To make optimal use of the processing power of multi-core machines, algorithms must be adapted.

This chapter discusses the techniques that we used to parallelize decision diagram algorithms in the parallel decision diagram library Sylvan [61, 64, 59]. These techniques are based on two main ingredients. The first ingredient is work-stealing to perform task-based algorithms such as decision diagram operations in parallel. The second ingredient consists of two concurrent data structures: a single shared hash table that stores all nodes of the decision diagrams, and a single concurrent operation cache that stores the intermediate results of operations for reuse.

This chapter is largely based on the research related to the parallel decision diagram library Sylvan, which is described in [66] and in the PhD thesis of Van Dijk [59]. Sylvan implements parallelized operations on binary decision diagrams (BDDs), list decision diagrams (LDDs), which are used in the model checking toolset LTSMIN [33], and multi-terminal binary decision diagrams (MTBDDs) [5, 22]. Sylvan can replace existing non-parallel implementations to bring the processing power of multi-core machines to non-parallel applications.

The remainder of this chapter is organized in the following way:

Section 2 gives a high-level overview of decision diagrams and decision diagram operations.

Section 3 discusses how decision diagram operations can be parallelized using work-stealing.

Section 4 discusses the main concurrent data structures: the hash table that contains the nodes of the decision diagrams, and the operation cache that stores the intermediate results of the operations.

Section 5 presents parallel garbage collection.

Section 6 briefly reviews the performance of parallel decision diagram operations for a number of applications. We discuss previously reported case studies on using decision diagrams in model checking, bisimulation reduction and probabilistic model checking.

Section 7 finally concludes the chapter.

2 Preliminaries

This section gives a high-level overview of decision diagrams and decision diagram operations. We discuss Boolean logic and the most well-known form of decision diagrams, binary decision diagrams, in Sections 2.1 and 2.2, as well as one popular extension of binary decision diagrams with non-binary leaves in Section 2.3. Section 2.4 describes how typical decision diagram operations are implemented. Section 2.5 discusses lock-free programming. Finally, Section 2.6 aims to provide the reader with an overview of parallelized decision diagram operations in earlier literature.

2.1 Boolean Logic and Notation

Boolean logic is fundamental in computer science, especially as all digital data can be expressed in binary form. Boolean variables are either `true` or `false`. Boolean formulas are defined on Boolean variables and have operators such as conjunction ($x \wedge y$), disjunction ($x \vee y$), negation ($\neg x$) and quantification ($\exists$ and $\forall$). Boolean functions are functions $\mathbb{B}^N \to \mathbb{B}$ (on N inputs), with a Boolean formula representing the relation between the inputs and the output of the Boolean function.

In this chapter, we also use 0 to denote `false` and 1 to denote `true`. We use the notation $f_{x=v}$ for a Boolean function f where the variable x is given value v. For example, given a function f defined on N variables:

$$f(x_1,\dots,x_i,\dots,x_N)_{x_i=0} \equiv f(x_1,\dots,0,\dots,x_N)$$
$$f(x_1,\dots,x_i,\dots,x_N)_{x_i=1} \equiv f(x_1,\dots,1,\dots,x_N)$$

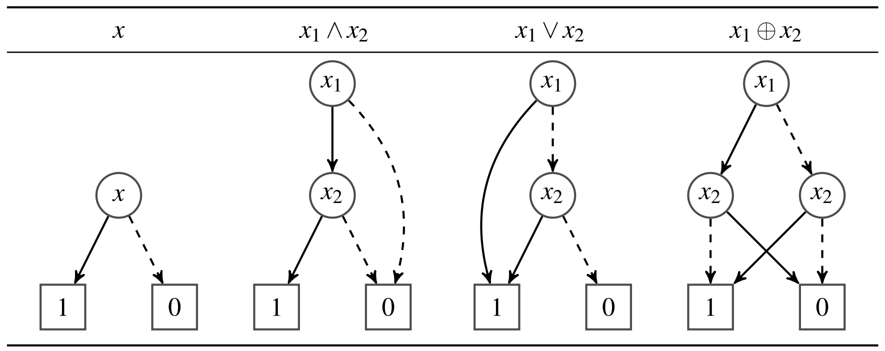
This notation is especially relevant for decision diagrams, as they are recursively defined on the value of a Boolean variable.

2.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions $\mathbb{B}^N \to \mathbb{B}$ [3, 14] and 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. Variables are encountered along each directed path according to a fixed variable ordering. Equivalent nodes (two nodes with the same label and outgoing edges) and nodes with two identical outgoing edges (redundant nodes) are forbidden. It is well known that, given a fixed ordering, every Boolean function is represented by a unique BDD [14].

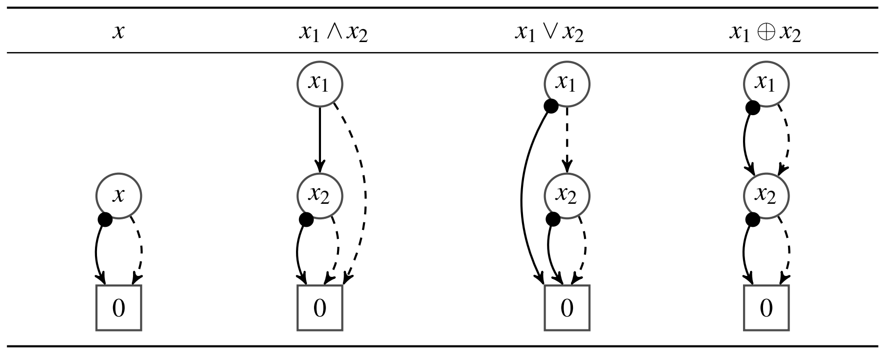
The following figure shows the BDDs for several Boolean functions. Internal nodes are drawn as circles with variables, and leaves as boxes. High edges are drawn solid, and low edges are drawn dashed. Given a valuation of the variables, BDDs are evaluated by following the high edge when the variable x is `true`, or the low edge when it is `false`.



There are various equivalent ways to interpret a binary decision diagram, leading to the same Boolean function:

1. Consider every distinct path from the root of the BDD to the terminal 1. Every such path assigns `true` or `false` to the variables encountered along that path, by following either the high edge or the low edge. In this way, every path corresponds to a conjunction of literals, sometimes called a cube. For example, the cube $x_0\overline{x_1}x_3x_4\overline{x_5}$ corresponds to a path that follows the high edges of nodes labeled x_0, x_3 and x_4, and the low edges of nodes labeled x_1 and x_5. If the cubes $c_1,\dots,c_k$ correspond to the k distinct paths in a BDD, then this BDD encodes the function $c_1 \vee \dots \vee c_k$.
2. Alternatively, after computing $f_{x=1}$ and $f_{x=0}$ by interpreting the BDDs obtained by following the high and the low edges, a BDD node with variable label x represents the Boolean function $xf_{x=1} \vee \overline{x}f_{x=0}$.

In addition, we use complemented edges [13] as a property of an edge to denote the negation of a BDD, i.e., the leaf 1 in the BDD will be interpreted as 0 and vice versa, or in general, each terminal node will be interpreted as its negation. This is a well-known technique. We write $\neg$ to denote toggling this property on an edge. The following figure shows the BDDs for the same simple examples as above, but with complemented edges:

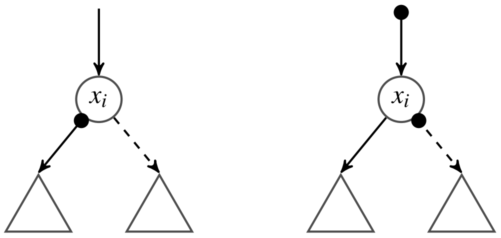


As this example demonstrates, always strictly fewer nodes are required, and there is only one ("false") terminal node. The terminal "true" is simply a complemented edge to “false”. We only allow complement marks on the high edges to maintain the property that BDDs uniquely represent Boolean functions (see also below).

The interpretation of a BDD with complemented edges is as follows:

1. Count the complemented edges on each path to the terminal 0. Since negation is an involution ($\neg\neg x = x$), each path with an odd number of complemented edges is a path to “true”, and with cubes $c_1, \dots, c_k$ corresponding to all such paths, the BDD encodes the Boolean function $c_1 \vee \cdots \vee c_k$.
2. 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 $x f_{x=1} \vee \overline{x} f_{x=0}$.

With complemented edges, the following BDDs are identical:



Complemented edges thus introduce a second representation of a Boolean function: if we toggle the complement mark on the two outgoing edges and on all incoming edges, we find that it encodes the same Boolean function. By forbidding a complement on one of the outgoing edges, for example the low edge, BDDs remain canonical representations of Boolean functions, since then the representation without a complement mark on the low edge is always used [13].

2.3 Multi-terminal Binary Decision Diagrams

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams (MTBDDs) have been proposed [5, 22] with arbitrary leaves, representing functions from the Boolean space $\mathbb{B}^N$ into any set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), floating-point numbers (encoding $\mathbb{B}^N \to \mathbb{R}$) or rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). In our implementation of MTBDDs, we also allow for partially defined functions, using a leaf $\perp$. See Figure 1 for a simple example of such an MTBDD.

Similar to the interpretation of BDDs, MTBDDs are interpreted as follows:

1. An MTBDD encodes functions from a Boolean domain $D \subseteq \mathbb{B}^N$ onto some codomain C, such that for each path to a leaf $V \in C$, all inputs matching the corresponding cube c map to V. Also, given all such cubes $c_1, \dots, c_k$, the domain D equals $c_1 \vee \cdots \vee c_k$. All paths corresponding to cubes not in D, i.e., for which the function is not defined, lead to the leaf $\perp$.

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)
$\exists v\colon x$	`exists`(x,v)
$\forall v\colon x$	`not(exists(not`(x)`,`v`))`

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

2. If an MTBDD is a leaf with the label V, then it represents the function $f(x_1,\ldots,x_N) \equiv V$. Otherwise, it is an internal node with label x. After recursively computing $f_{x=1}$ and $f_{x=0}$ by interpreting the MTBDDs obtained by following the high and the low edges, the node represents a function $f(x_1,\ldots,x_N) \equiv$ if x then $f_{x=1}$ else $f_{x=0}$.

Like BDDs, MTBDDs can have complement edges. This works only for leaf types for which negation is properly defined, i.e., each leaf x has a unique negated counterpart $\neg x$, such that $\neg\neg x = x$ and $\neg x \neq x$. In general, this does not work for numbers as $0 = -0$ in ordinary arithemetic. In addition, this also does not work for partially defined functions, as the negation of $\perp$ is not properly defined. In practice this means that complement edges are not typically used with MTBDDs.

2.4 Algorithms on Decision Diagrams

Many BDD packages implement the basic BDD operations `and`, `not` and `xor`, the if-then-else (`ite`) operation, and `exists` (Table 1). Negation $\neg$ is performed

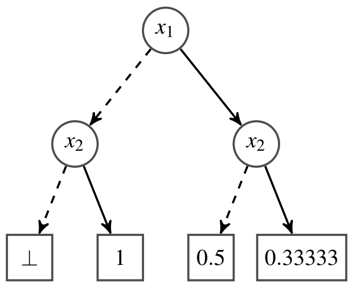


Fig. 1 A simple MTBDD for a function which maps $\overline{x_1}x_2$ to 1, $x_1\overline{x_2}$ to 0.5 and x_1x_2 to 0.33333. The function is undefined for the input $\overline{x_1}\overline{x_2}$

Algorithm 1: The BDD algorithm `and`, with the BDDs x and y as parameters. The result is a BDD representing $x \wedge y$

```
def and(x, y):
    if $x = 1$ : return $y$
    if $y = 1 \vee x = y$ : return $x$
    if $x = 0 \vee y = 0 \vee x = \neg y$ : return 0
    if $x > y$ : swap $x$ and $y$
    if result $\leftarrow$ cache$[(x, y)]$ : return result
    $v \leftarrow$ topvar$(x,y)$
    low $\leftarrow$ and$(x_{v=0}, y_{v=0})$
    high $\leftarrow$ and$(x_{v=1}, y_{v=1})$
    result $\leftarrow$ lookupBDDnode$(v$, low, high$)$
    cache$[(x, y)] \leftarrow$ result
    return result
```

using complemented edges (Section 2.2) and is basically free. See Algorithm 1 for a typical implementation of `and`.

This algorithm showcases all features of a typical decision diagram operation. Most decision diagram operations first check whether the operation can be applied immediately to x and y (lines 2–4). This is typically the case when x and y are leaves. Often there are also other trivial cases that can be checked first. In Algorithm 1, this is the case when $x = y$ or when $x = \neg y$.

Often, the parameters of an operation can be normalized in some way to increase the cache efficiency. For example, $a \wedge b$ and $b \wedge a$ are the same operation. Normalization rules can then rewrite the parameters to some standard form in order to increase cache utilization, as at line 5. A well-known example is the if-then-else algorithm, which rewrites using rewrite rules called "standard triples" as described in [13].

We consult the operation cache (line 6) to see whether this (sub)operation has been computed earlier. The operation cache is required to reduce the time complexity of BDD operations from exponential to polynomial in the size of the BDDs.

If x and y are not leaves and the operation is not trivial or in the cache, we use a function `topvar` (line 7) to determine the first variable of the root nodes of x and y. If x and y have different variables in their root nodes, `topvar` returns the first one in the variable ordering of x and y. We then compute the recursive application to the cofactors of x and y with respect to variable v at lines 8–9.

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 has 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 10). This is done by a function `lookupBDDnode`, which, given a variable v and the BDDs of result$_{v=0}$ and result$_{v=1}$, returns the BDD for result by consulting the unique table.

When the result has been computed, we store it in the operation cache (line 11) and return the result (line 12).

2.5 Parallelism

A major goal in computing is to perform ever larger calculations and to improve their performance and efficiency. This can be accomplished using various techniques that are often orthogonal to each other, such as better algorithms, faster processors and parallel computing using multiple processors. Faster hardware increases the performance of most computations, often regardless of the algorithm, although some algorithms benefit more from processor speed while others benefit more from faster memory access. For suitable algorithms, parallel processing can considerably improve the performance, on top of what is possible just by increased processor speeds.

For some algorithms, efficient parallelism is almost trivial. It is no coincidence that graphics cards contain thousands of small processors, resulting in massive speedups for very particular applications. Other algorithms are more difficult to parallelize. For example, some algorithms are inherently sequential, with few opportunities for the parallel execution of independent calculation paths. Other algorithms have enough independent paths for parallelization in theory, but are difficult to parallelize in practice, for example because they are irregular and continually require load balancing, moving work between processors. Some algorithms are memory-intensive, i.e., they spend most of their time manipulating data in memory, which can result in bottlenecks due to the limited bandwidth between the processors and the memory, as well as time spent waiting in locks.

This chapter discusses the parallelization of algorithms for decision diagrams, which are large directed acyclic graphs. They are typically irregular and mainly consist of unpredictable memory accesses with high demands on memory bandwidth. Decision diagrams are often used as the underlying operations of other algorithms. If the underlying decision diagram operations are parallelized, then sequential algorithms that use them may also benefit from the parallelization.

Lock-free programming

In parallel programs, memory accesses can result in race conditions or data corruption, for example when multiple threads write to the same location in memory. Typically data structures are protected against race conditions using locking techniques. While locks are relatively easy to implement and reason about, they often severely cripple parallel performance, especially as the number of threads increases. Threads have to 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 data structures 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.

```
def compare-and-swap(location, expected, newvalue):
    value ← *location
    if value $\neq$ expected : return False
    *location ← newvalue
    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.

Data structures that avoid locks are called non-blocking or lock-free. Such data structures 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.

2.6 Historical Perspective

This section describes various approaches have been tried in the past for parallel processing of decision diagrams, as discussed in [59].

Massively parallel computing (early 1990s)

In the early 1990s, researchers tried to speed up BDD manipulation by parallel processing. The first paper [34] views BDDs as automata, and combines them by computing a product automaton followed by minimization. Parallelism arises by handling independent subformulas 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 [46] or massively parallel SIMD machines [17, 28] with up to 64K processors. Experiments were run on supercomputers, such as the Connection Machine. Given the large number of processors, the speedup (around 10 to 20) was disappointing.

Parallel operations and constructions

An interesting contribution in this period is the paper by Kimura et al. [35]. Although they focus on the construction of BDDs, their approach relies on the observation that suboperations of 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 two years later [10]. Similarly to [35], 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 implement the parallel construction of BDDs for logic circuits using lock-based distributed hash tables, parallelizing on the structure of the circuits [18]. Yang and O’Hallaron [71] parallelize breadth-first BDD construction on multi-processor systems, resulting in reasonable speedups of up to $4\times$ with eight processors, although there is a significant synchronization cost due to their lock-protected unique table.

Distributed memory solutions (late 1990s)

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 [4, 58, 7] and breadth-first [54] 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 machine, but that their memory can be combined in order to handle larger BDDs. For example, even though [58] reports a nice parallel speedup, the performance with 32 machines is still $2\times$ slower than the non-parallel version. BDDNOW [43] 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 [29, 19] or shared memory [23, 21]. Here, BDD partitioning strategies such as horizontal slicing [19] and vertical slicing [31] were used to distribute the BDDs over the different computers. Also the saturation algorithm [20], an optimal iteration strategy in symbolic reachability, was parallelized using horizontal slicing [19] and using the work-stealer Cilk [23], although it is still difficult to obtain good parallel speedup [21].

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 [39] allows execution of BDD operations from multiple threads, but does not parallelize single BDD operations. Similarly, the well-known sequential BDD implementation CUDD [57] 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 [62, 61, 64] 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 et al. [47, 48] continued our work by parallelizing BDD operations on shared-memory abstractions of distributed systems using remote direct memory access. Work by Velev et al. [68] implements BDD operations on GPUs for a small case study with promising results.

3 Parallel Decision Diagrams

The requirements for the efficient parallel implementation of decision diagrams are not the same as for a non-parallel implementation. We refer to Somenzi [56] for a general discussion on the implementation of non-parallel decision diagrams. Somenzi already established several aspects of a BDD package. The two central data structures of a BDD package are the *unique table* (or *nodes table*) and the *computed table* (or *operation cache*). Furthermore, garbage collection is essential for a BDD package, as most BDD operations continuously create and discard BDD nodes. The two central data structures are discussed in Section 4 and garbage collection in Section 5. The current section presents the parallelization of decision diagram operations by work-stealing.

3.1 Work-Stealing

Operations on decision diagrams are typically recursively defined on the structure of the inputs. To parallelize decision diagram operations, we consider each subproblem as a separate task and execute independent tasks in parallel. This type of parallelism is called *task-based parallelism*.

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 effec-

Algorithm 2: The algorithm (left) is implemented (right) using `SPAWN`, `SYNC` and `CALL`

```
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
```

tive load-balancing method [10], with implementations such as Cilk [11, 27] and Wool [24, 25] that allow parallel programs to be written in a style similar to sequential programs [2]. Work-stealing has been proven to be optimal for a large class of problems and has tight memory and communication bounds [10].

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. Worker are idle either because they do not have any tasks to perform (e.g., at the start of a computation), or because all their subtasks have been stolen and they have to wait for the result of the stolen subtasks 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 Algorithm 2. 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.

One important aspect of the work-stealing algorithm is victim selection. For example in systems with hierarchy, e.g., a network of workstations, it might be useful to steal from local workers first before trying to steal from a remote worker. Another strategy would be to remember how much work other workers have after a steal attempt, and use this to intelligently select targets. In our implementation, workers with an empty task pool steal from random victims.

When synchronizing with a stolen task, a possible strategy for the victim is to steal from the thief until the stolen task is completed. By stealing back from the thief, a worker executes subtasks of the stolen task. This technique is called leapfrogging [69]. When stealing from random workers instead, the size of the task pool of each worker could grow beyond the size needed for complete sequential execution [25], since stealing will build a new stack on top of the blocked join. Using leapfrogging rather than stealing from random workers thus limits the space requirement of the task pools to that of sequential execution, although in practice it is expensive to guarantee that the tasks that are stolen from the thief are really subtasks of the original task. It might be possible that the thief finished the original task and

Work-stealing operations	Task pool operations
`spawn`(task)	`push`(task)
`sync`	`peek`, `pop`
`steal-and-run`(victim)	`steal`

Table 2 Operations of the work-stealing algorithm and matching operations of the task pool of each worker

stole a different branch of the task tree after the victim checked the status of the stolen task. Our implementation also uses the leapfrogging strategy.

Another concern is which task(s) to steal. A simple algorithm is to steal the first unstolen task from the bottom of the stack. A variation could be to steal multiple tasks, or to steal a random task from anywhere in the stack. In our implementation, thieves steal the first unstolen task from the bottom of the stack.

See Table 2 for an overview of the work-stealing operations and how they match with operations on the task pool. The methods `spawn` and `sync` implement the keywords `SPAWN` and `SYNC`. The method `steal-and-run` tries to steal a task from the given victim and, if successful, executes the task and communicates the result back to the owner of the task. The methods `push`, `peek`, `pop` and `steal` are implemented by the task pool:

- The `push`, `peek` and `pop` operations are only used by the owner of the stack, and the `steal` operation only by thieves.
- The `push` operation puts a task on the stack.
- The `peek` operation fixes the status of the task at the top of the stack: either stolen or available as work. After `peek`, the top task, if not stolen, cannot be stolen until the next `push` (or if `peek` is called again).
- The `pop` operation removes the topmost task from the stack. Furthermore we assume that the task data remains in the task pool until overwritten by a `push` operation.
- The `steal` operation steals a task from the bottom of the stack, changing its status from available work to stolen work. Stolen tasks are kept on the stack so the results of tasks can be communicated back to the original owner of the task.

Different implementations of the work-stealing stack can be used, as long as they implement the described functionality. Experiments show that the difference in performance between the private deque by Acar et al. [2], the shared deque in Wool [24, 25] and the shared deque we implemented in Lace [63] are relatively small; they all have sufficient scalability, although Lace also implements a stop-the-world feature required for garbage collection (Section 5).

Algorithm 3: The implementation of work-stealing using leapfrogging when waiting for a stolen task to finish, i.e., steal from the thief

```
def spawn(task):
    push(task)

def sync():
    res ← peek()
    // res is Work(task) or Stolen(task)
    if res = Work(task) :
        pop()
        return task.execute()
    else:
        while task.thief = None : (loop)
        while ¬ task.done : steal-and-run(task.thief)
        pop-stolen()
        return task.result

def steal-and-run(victim):
    if victim.steal() = Task(stolentask) :
        stolentask.thief ← me
        result ← stolentask.execute()
        stolentask.result ← result
        stolentask.done ← True

thread worker(id, roottask):
    done ← False
    if id = 0 :
        roottask.execute()
        done ← True
    else: while done is False: steal-and-run(random victim)
```

3.2 Parallel Operations with Work-Stealing

Decision diagram operations such as `and` (Algorithm 1) are parallelized by executing the subtasks (lines 8–9) in parallel:

```
8  do in parallel:
9      low ← and($x_{v=0}$, $y_{v=0}$)
10     high ← and($x_{v=1}$, $y_{v=1}$)
```

This is equivalent to the following:

```
8  SPAWN(and, $x_{v=0}$, $y_{v=0}$)
9  high ← CALL(and, $x_{v=1}$, $y_{v=1}$)
10 low ← SYNC
```

A more involved example is the parallelized algorithm `exists` (Algorithm 4), which computes existential quantification. This algorithm receives the input param-

Algorithm 4: Parallelized BDD algorithm `exists`, with the BDD x and V the cube of variables that are abstracted via existential quantification

```
1  def exists($x, V$):
2      if $x = 0 \vee x = 1 \vee V = \emptyset$ : return $x$
3      $v$ = var($x$)
4      while $V \neq \emptyset \wedge$ var($V$) $< v$ : $V \leftarrow$ next($V$)
5      if $V = \emptyset$ : return $x$
6      if result $\leftarrow$ cache$[(x,V)]$ : return result
7      if $v =$ var($V$) :
8          if $x_{v=0} = 1 \vee x_{v=1} = 1 \vee x_{v=0} = \neg x_{v=1}$ : result $\leftarrow 1$
9          else:
10             low $\leftarrow$ exists($x_{v=0}$, next($V$))
11             if low $= 1$ : result $\leftarrow 1$
12             else:
13                 high $\leftarrow$ exists($x_{v=1}$, next($V$))
14                 result $\leftarrow$ or(low, high)
15     else:
16         do in parallel:
17             low $\leftarrow$ exists($x_{v=0}$, $V$)
18             high $\leftarrow$ exists($x_{v=1}$, $V$)
19         result $\leftarrow$ lookupBDDnode($v$, low, high)
20     cache$[(x,V)] \leftarrow$ result
21     return result
```

eters x and V, where x is the BDD representing the function to which quantification is applied, and V is the BDD representing the conjunction of the variables that are abstracted away from x. After the trivial cases (line 2), we check whether V actually contains variables that are in the BDD (lines 3–5), exploiting the fact that V is also an ordered BDD. This is also a normalization step for the cache, which is checked at line 6. Now, there are two cases: either the current root variable v is in V (lines 7–14) or it is not in V (lines 15–19). In the second case, we simply perform the two suboperations in parallel and compute the result. In the first case, after checking some trivial cases, we can either 1) perform the two suboperations in parallel; 2) perform the "low" suboperation first; or 3) perform the "high" suboperation first. If either of these suboperations returns 1, then the other does not need to be computed. The advantage of option 1 is that there is more opportunity for parallelization, at the cost of possible extra work. However, this extra independent work might not be necessary, since there is already a lot of independent work from the parallelization at lines 17–18 and inside the `or` operation. In Algorithm 4, we compute the "low" suboperation first.

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 $\mathbf{x} = x_1, x_2, \ldots, x_N$. Transitions between these states are represented using Boolean variables $\mathbf{x}$ for the source

Algorithm 5: 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 $\mathbf{x} \cup \mathbf{x}'$) computes the set of successor states defined on $\mathbf{x}$, i.e., $\big(\exists \mathbf{x}\colon\ (S \wedge R)\big)[\mathbf{x}' := \mathbf{x}]$. We assume that all variables in R are also in V

```
def relnext ($S, R, V$):
    if $S = 0 \vee R = 0$ : return 0
    if $S = 1 \wedge R = 1$ : return 1
    $v$ = topvar ($S$,$R$)
    while var ($V$) $< v$ : $V \leftarrow$ next ($V$)
    // if $V = \emptyset$, we assume $R$ is irrelevant
    if $V = \emptyset$ : return $S$
    if result $\leftarrow$ cache[$(S, R, V)$] : return result
    if $v =$ var ($V$) :
        x, x' $\leftarrow$ unprimed $v$, primed $v$
        V' $\leftarrow$ V without x and x'
        do in parallel:
            a $\leftarrow$ relnext ($S_{x=0}$, $R_{x=0,x'=0}$, $V'$)
            b $\leftarrow$ relnext ($S_{x=1}$, $R_{x=1,x'=0}$, $V'$)
            c $\leftarrow$ relnext ($S_{x=0}$, $R_{x=0,x'=1}$, $V'$)
            d $\leftarrow$ relnext ($S_{x=1}$, $R_{x=1,x'=1}$, $V'$)
        do in parallel:
            low $\leftarrow$ or (a, b)
            high $\leftarrow$ or (c, d)
        result $\leftarrow$ lookupBDDnode ($x$, low, high)
    else:
        // $v$ is not in $R$, by assumption
        do in parallel:
            low $\leftarrow$ relnext ($S_{v=0}$, $R$, $V$)
            high $\leftarrow$ relnext ($S_{v=1}$, $R$, $V$)
        result $\leftarrow$ lookupBDDnode ($v$, low, high)
    cache[$(S, R, V)$] $\leftarrow$ result
    return result
```

states and variables $\mathbf{x}' = x'_1, x'_2, \ldots, x'_N$ for the target states. Given a set of states S_i encoded as a BDD on variables $\mathbf{x}$, and a transition relation R encoded as a BDD on variables $\mathbf{x} \cup \mathbf{x}'$, the set of states S'_{i+1} encoded on variables $\mathbf{x}'$ is obtained by computing $S'_{i+1} = \exists \mathbf{x}\colon\ (S_i \wedge R)$. 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 successor states defined on the unprimed variables $\mathbf{x}$ instead of the primed variables $\mathbf{x}'$, so the `and_exists` call is then followed by a variable substitution that replaces all occurrences of variables from $\mathbf{x}'$ with the corresponding variables from $\mathbf{x}$. 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. This combination of `and_exists` and variable renaming can be done with a specialized operation `relnext`, which computes the successors of sets of states, where the transition relation is encoded with the interleaved variable ordering.

See Algorithm 5 for the parallel 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 $\mathbf{x}$ and $\mathbf{x}'$ (the variables on which the transition relation is defined). We first check for terminal cases (lines 2–3). These are the same cases as for the $\wedge$ 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 transitions (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).

3.3 Conclusion

This section discussed using work-stealing to perform operations on decision diagrams in parallel. We looked at three operations in particular: `and`, which is a prototype for many simple decision diagram operations; `exists`, which adds the complexity that the subtasks are not completely independent (if “low” returns 1, “high” does not need to be computed); and `relnext`, which adds the complexity of having two phases with independent subtasks.

4 Concurrent Data Structures

To efficiently parallelize decision diagram operations, we must perform memory operations in a scalable manner, i.e., using optimized scalable data structures. This section describes the organization of decision diagram nodes in memory, as well as the design of the unique table and the operation cache.

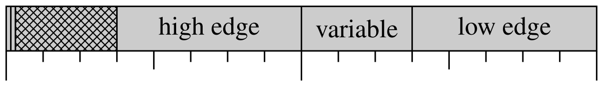
4.1 Representation of Nodes

The representation of BDD and MTBDD nodes in memory is important for both the sequential and the parallel performance of decision diagram implementations. 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. With 16 bytes per node, exactly four nodes fit in a cacheline of 64 bytes (the size of the cacheline for many current computer architectures, in particular the x86 family that we use). If the unique table is properly aligned in memory, then only one cacheline needs to be accessed when accessing a node.

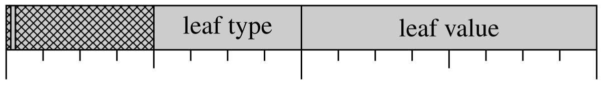
We use 40 bits to store the index of a node in the unique table. This is sufficient to store up to 2^{40} nodes, i.e., 16 terabytes of nodes, excluding overhead costs.

Sylvan defines the type MTBDD as a 64-bit integer, representing an edge to an MTBDD node. The lowest 40 bits represent the location of the node in the nodes table, and the most significant bit stores the complement mark [13], mainly used by BDDs. The BDD 0 is reserved for the leaf `false`, with the complemented edge to 0 (i.e., `0x8000000000000000`) meaning `true`.

Internal BDD and 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):



MTBDD leaves store the leaf type (32 bits), the leaf value (64 bits) and the fact that they are a leaf (1 bit, set to 1):



The unused space bits are set to 0. They can also be used by the decision diagram library for other node types or for temporary marking of nodes in algorithms, which is beyond the scope of this chapter.

4.2 Unique Table

The unique table stores all decision diagram nodes and is essential to avoid duplicate nodes. This table is typically implemented as a hash table, in particular because the `find-or-insert` operation is performed in time O(1) on average (amortized) by a hash table.

The unique table can either be one shared table, or be split into multiple parts somehow. For example, Somenzi [56] argues for a subtable for each variable level, as this makes the implementation of variable reordering easier. The disadvantage of subtables is that their sizes must be adjusted dynamically, thus requiring the different parallel processes to cooperate on performing garbage collection and resizing when subtables are full. In addition, there is some overhead to compute the correct size for each table, which can be avoided by using a single table. Finally, subtables require the additional complexity of decreasing subtable sizes and compressing decision diagrams, which we avoid by using a single table that only increases in size when this is needed.

In the past, there have been various proposals to split the unique table into several parts for parallel applications, for example to assign parts of the decision diagrams to certain processors or workstations. This is a consideration that can be orthogonal to parallelism. As we use work-stealing to perform the load balancing of the decision diagram operations, we have no control over which processor performs specific operations. Therefore, we use a single continuous block of memory, and we let the operating system take care of allocating memory blocks on all available memories in the system.

The unique table essentially requires the following operations, which must be highly scalable:

- a `find-or-insert` method, which, given a 16-byte node, either finds the existing node in the table, or creates a new node.
- a method to delete nodes for garbage collection. Our implementation has a separate "data array" containing the nodes and a "hash array" containing the metadata. We require three operations:
  - `clear` removes all entries from the hash array;
  - `mark` marks a given node for reinsertion in the hash array; and
  - `rehash` reinserts a given node in the hash array.

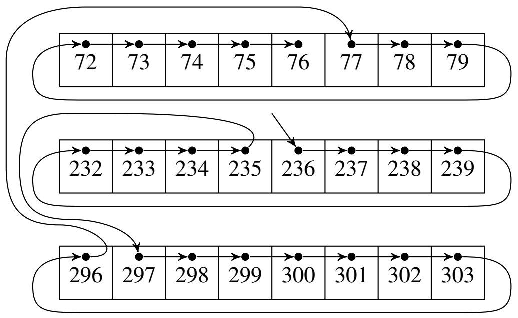
Our design strictly separates lookup and insertion of nodes from a stop-the-world garbage collection phase, during which the table may be resized. From the perspective of the nodes table algorithms (and correctness), all threads of the program are in one of two phases:

1. During *normal operation*, threads only call the `find-or-insert` operation, which takes as input the 16-byte data 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*, the `find-or-insert` operation is never called. Instead, methods `clear`, `mark` and `rehash` (described in Section 5) are called to perform garbage collection.

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 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.

The unique table we use in Sylvan is based on the hash table in [36], which is designed to store visited states in model checking. This hash table incorporates two ideas that we also use in our design:

- Using a probe sequence called "walking-the-line" that is efficient with respect to transferred cachelines.
- Separating the stored data in a "data array" and the hash of the data in the "hash array" to avoid directly comparing the data.



Order of buckets:
236–239, 232–235,
297–303, 296,
77–79, 72–76

Fig. 2 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

Furthermore, to manage the "data array" we use bit arrays as a convenient parallel allocator, although other scalable parallel allocation mechanisms for fixed-size (16 bytes) memory blocks could be used to manage the data array.

The walking-the-line 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, to either detect that the data is already in the hash table, or to find an empty bucket, which indicates that the data can be inserted into that bucket. 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 [36]. 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. For example, if there are eight buckets per cacheline and we assume that the buckets are properly aligned so that the first cacheline starts with bucket 0, then linear probing starting at bucket 61 would only check buckets 61–63 of the first accessed cacheline. In walking-the-line, the other buckets in that cacheline are also checked, so after buckets 61–63, also buckets 56–60 would be checked. Then, a new hash value is obtained for the data using a hash function to obtain the next starting bucket. In theory, this procedure could be repeated forever; in practice, after a certain number of cachelines the procedure terminates with the result that the table is full. See also Figure 2 for an example of walking-the-line.

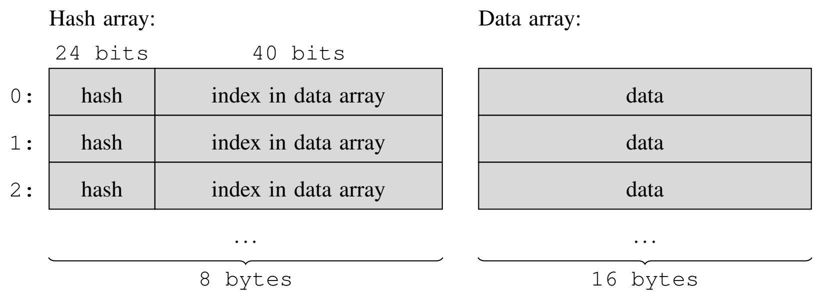


Fig. 3 Layout of the hash array and data array

Separated arrays

The hash table stores the hash of the data in each bucket in a separate array. 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`.

Bit arrays for data management

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`. We therefore only use 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 an initial region using the `claim-next-region` method in Algorithm 6. 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.

The layout of the hash array and the data array is given in Figure 3. We use a hash function that never hashes to 0 and we forbid nodes with the index 0 because 0

Algorithm 6: Algorithm for parallel `find-or-insert` of the hash table, with 512 buckets per region. The variable `myregion` is a thread-specific variable

```
def find-or-insert(data):
    index ← 0
    h ← hash(data)
    for s ∈ probe-sequence(data) :
        V ← harray[s]
        if V = 0 :
            if index = 0 :
                index ← reserve-data-bucket()
                darray[index] ← data
            if cas(harray[s], 0, {h, index}) : return index
            else: V ← harray[s]
        if V.hash = h ∧ darray[V.index] = data :
            if index ≠ 0 : free-data-bucket(index)
            return V.index
    raise TableFull
def reserve-data-bucket():
    loop:
        if myregion has a bit set to 0 :
            i ← first bit in myregion that is 0
            set-bit(databits, 512 × myregion + i, 1)
            return 512 × myregion + i
        else: myregion ← claim-next-region(myregion)
def free-data-bucket(d):
    set-bit(databits, d, 0)
def claim-next-region(oldregion):
    newregion ← (oldregion + 1) mod (tablesize/512)
    while newregion ≠ oldregion :
        loop:
            if the bit for newregion is 1 : break
            if set-bit-cas(regionbits, newregion, 0, 1) : return newregion
        newregion ← (newregion + 1) mod (tablesize/512)
    raise TableFull
```

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 tries to find 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 the `reserve-data-bucket` function. If the current region is full, then the thread claims a new region with the `claim-next-region` function. Note that it may be possible that the next region contains used buckets, if there has been a garbage collection earlier. Afterwards the new bucket is 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 6. 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 into the data array. For every bucket in the probe sequence, we first check whether 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 succesful, 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 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 `reserve-data-bucket` and on `free-data-bucket`, which are also given in Algorithm 6. They are fairly straightforward.

The `claim-next-region` method searches in the `regionbits` array for the first 0-bit. 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.

4.3 Computed Table

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 [56]. As with the unique table, we use only one shared operation cache for all operations, because we want to minimize interaction between workers, such as synchronization when shared parts of memory are resized.

In [56], Somenzi writes that a lossless computed table guarantees polynomial cost for the basic synthesis operations, but that lossless tables (which do not throw away results) are not feasible when manipulating many large BDDs and in practice lossy computed tables (which may throw away results) are implemented. If the cost

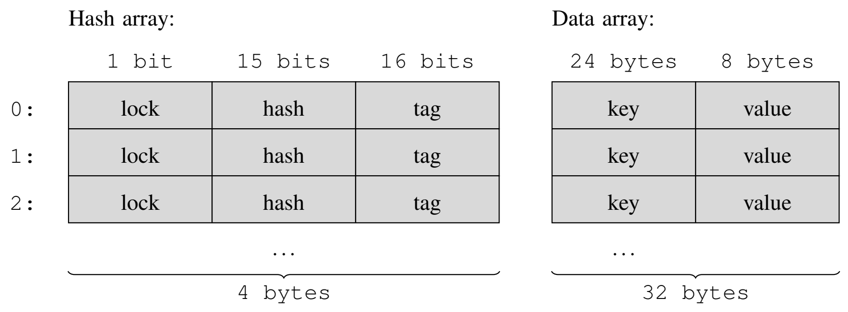


Fig. 4 Layout of the operation cache

of recomputing subresults is sufficiently small, it can pay to regularly delete results or even prefer to sometimes skip the cache to avoid data races. We design the operation cache to abort operations as early as possible when there may be a data race or the data may already be in the cache.

We use an operation cache that consists of two arrays: the hash array and the data array. See Figure 4 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 (see Section 4.3) 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 "tag" counter that increments (modulo 4096) with every update to the bucket, and check this value before reading the cache and after reading the cache to check that the obtained result is valid. The chance that this tag counter is the same for a different result is astronomically small, as this requires exactly 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 the same hash value but different data.

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 one cacheline transfer.

See Algorithms 7 and 8 for the `cache-put` and `cache-get` algorithms.

The algorithms are quite straightforward. 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

Algorithm 7: The `cache-put` algorithm

```
def cache-put (key, value):
    h, location ← hash(key)
    s ← harray[location]
    if s.lock : return
    if s.hash = h : return
    if not cas(harray[location], s, $\{1, h, s.tag + 1\}$) : return
    darray[location] ← $\{key, value\}$
    harrray[location] ← $\{0, h, s.tag + 1\}$
```

Algorithm 8: The `cache-get` algorithm

```
def cache-get (key):
    h, location ← hash(key)
    s ← harray[location]
    if s.lock : return ⊥
    if s.hash ≠ h : return ⊥
    storedkey, value ← darray[location]
    if storedkey ≠ key : return ⊥
    if s ≠ harray[location] : return ⊥
    return value
```

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).

It is theoretically 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, with at least one of these such that the comparison at line 7 succeeds. 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.

5 Garbage Collection

Operations on decision diagrams typically create many new nodes and discard old nodes. Nodes that are no longer referenced are typically 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 [56].

There are various approaches to garbage collection. For example, a *reference count* could be added to each node, which records how often the 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 should be kept and removes all unmarked nodes. We refer to [56] 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 when there is only one processor, but with many processors this results in excessive communication, especially for nodes that are commonly 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 allowing some workers to continue inserting nodes during garbage collection.

Some decision diagram implementations maintain a counter 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” (see Section 4) 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 in Section 4, 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 approach of parallel garbage collection consists of the following steps:

1. Initiate the operation using the work-stealing framework (e.g., as supported by Lace) to arrange the “stop-the-world” interruption of all ongoing tasks. This feature is described below.
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 would also be possible.
3. Mark all nodes that we want to keep, allowing 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 all marked nodes in the hash array of the unique table.

The garbage collection process itself is also executed in parallel using task parallelism. 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. 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.

Various mechanisms can be used to store the set of nodes to be kept in step 3. Operations must often temporarily store subresults that may not be removed; we use thread-local stacks to store these subresults, which minimizes worker interactions. External references (outside of operations) are less sensitive to these interactions; one can use any kind of set implementation (we use a simple hash table) to implement this; an important optimization is to not store references to nodes directly, but pointers to the variables; this way, updating a variable does not incur calls to remove and add references.

One helpful feature for garbage collection in Sylvan that we implemented in the work-stealing framework 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 exact implementation depends on the queue and involves several steps, where workers regularly check a flag in shared memory and use barriers to coordinate starting a new task tree. Further details are beyond our scope here, as they strongly depend on the used queue implementation. Interested readers are referred to [59].

6 Empirical Results

This section showcases the performance of parallel decision diagram operations in a number of applications, as reported in the literature. We briefly introduce model checking using decision diagrams in Section 6.1. We show the performance for symbolic on-the-fly reachability in the LTSMIN toolset as discussed in [62, 61, 64, 33, 59] in Section 6.2. For symbolic bisimulation minimization, which is related to symbolic model checking, we obtained good performance results in [65], which we report in Section 6.3. Finally, in Section 6.4 we discuss a performance comparison with other decision diagram implementations [60], showing that decision diagrams can be parallelized effectively without much overhead.

6.1 Symbolic Model Checking

As modern society increasingly depends on automated and complex systems, the safety demands on such systems increase as well. We depend on automated systems for basic infrastructure, to clean our water, to supply energy, to control our cars and trains, to monitor and process our financial transactions and for the internet. We use systems for entertainment when watching TV or using the phone, or for cooking with modern stoves, microwaves and fridges. Failure or unexpected behavior in these ubiquitous systems can have many consequences, from mild annoyances to fatal accidents. This motivates research into the formal verification of such systems, as well as computing properties such as failure rates and time to recovery.

In model checking, systems are modeled as sets of possible states of the system and transitions between these states. System states are typically represented by Boolean vectors. Fixed-point algorithms, which are procedures that repeatedly apply some operation until a fixed point is reached, play a central role in many model checking algorithms. An example of a fixed-point algorithm is state space exploration (“reachability”), which computes all states reachable from the initial state of the system. Many model checking algorithms depend on state space exploration to determine the number of states, to check whether an invariant is always true, to find cycles and deadlocks, and so forth.

A major challenge in model checking is that the space and time requirements of these algorithms increase exponentially with the size of the models. One technique to alleviate this problem is symbolic model checking [15, 16]. Symbolic model checking operates on sets of states and transitions, rather than individual states and transitions. These sets are then represented by their characteristic (Boolean) functions, which can be stored using BDDs. One advantage of using BDDs for fixed point computations is that equivalence testing is a trivial check, since BDDs uniquely represent Boolean functions. As small Boolean formulas can describe very large state spaces, symbolic model checking has been very successful at pushing the limits of model checking in the past [15]. Symbolic representations are also quite natural for the composition of multiple transition systems, e.g., when composing systems from subsystems.

Experiment	T_1	T_{48}	T_1/T_{48}
`firewire_link.1`	4.24	0.48	8.8
`anderson.1`	8.93	6.21	1.4
`firewire_tree.1`	4.23	0.30	14.1
`blocks.4`	635.86	17.27	36.8
`collision.5`	341.57	10.99	31.1
`lifts.8`	416.04	13.05	31.9
`exit.4`	494.85	13.95	35.5
`telephony.8`	915.61	28.18	32.5
Sum of all 269 models	16231	896	18.1

Table 3 Benchmark results (runtimes in seconds) for symbolic on-the-fly reachability with the LTSMIN toolset. Each data point is the average of at least five measurements

6.2 Symbolic On-the-Fly Reachability

LTSMIN is a model checking toolset that provides a language-independent Partitioned Next-State Interface (PINS), which connects various input languages to model checking algorithms [9, 37, 62, 33, 42]. 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," further distinguished by the variables that are "read" and the variables that are "written" [42]. This enables the efficient encoding of transitions that only affect some integers of the state vector. Exploiting this information lets the PINS interface work in a quasi-symbolic way, as a single pair of short vectors can represent many transition relations on the full state vector. Initially, LTSMIN does not have knowledge of the transitions in each transition group, and only the initial state is known. The transition system is explored by learning new transitions via the PINS interface, which are then added to the transition relation.

We evaluated the application of parallelization to LTSMIN [64, 59]. The experimental evaluation was based on the BEEM model database [51]. We performed the benchmarks on 269 benchmark models on a 48-core machine, consisting of four AMD Opteron$^{\text{TM}}$ 6168 processors with 12 cores each and 128 GB of internal memory. A summary of results is given in Table 3.

As is clear from these results, obtained speedups (T_1/T_{48}) strongly depend on the models; for some models, we obtain speedups above 30×, up to 36.8× for the `blocks.4` model.

See Figure 5 for a speedup graph of a selection of these models. This speedup graph was obtained using list decision diagrams, which are discussed in [59] and are beyond the scope of this chapter. The speedup graph suggests that most likely further speedups would be obtained after 48 cores for the selected models.

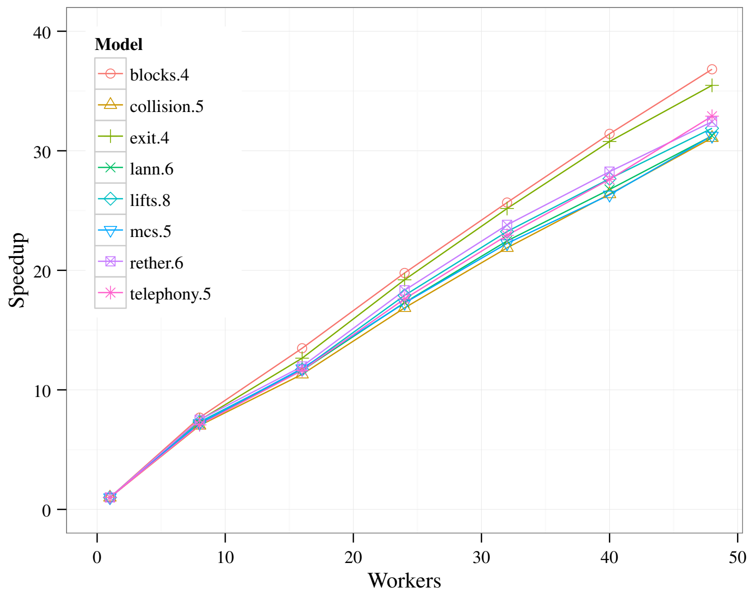


Fig. 5 Speedup graphs of several well-performing models. Each data point is an average of at least five measurements

6.3 Symbolic Bisimulation Minimisation

One of the main challenges for model checking is that the space and time requirements of model checking algorithms increase exponentially with the size of the models. One technique that helps combat this challenge is called bisimulation minimization. Given an input model, bisimulation minimization computes the smallest equivalent model, also called the maximal bisimulation, under some notion of equivalence. This can significantly reduce the number of states. This technique is also used to abstract models from internal behavior, when only observable behavior is relevant.

The maximal bisimulation of a model is typically computed using partition refinement. Starting with an initially coarse partition (e.g., all states are equivalent), the partition is refined until states in each equivalence class can no longer be distinguished. The result is the maximal bisimulation with respect to the initial partition. Blom et al. [8] introduced a signature-based method, which assigns states to equivalence classes according to a characterizing signature. This method easily extends to various types of bisimulation.

In [65, 67], we studied bisimulation minimization for labeled transition systems (LTSs), continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs), which combine the features of LTSs and CTMCs. These allow the analysis

SIGREF LTS models			Signature refinement			Quotient computation		
Model	States	Blocks	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
kanban03	1024240	85356	10.09	0.88	11.52×	6.72	0.35	19.08×
kanban04	16020316	778485	148.15	11.37	13.03×	106.22	5.38	19.73×
kanban05	16772032	5033631	1284.86	73.57	17.47×	740.53	33.80	21.91×
SIGREF CTMC models			Signature refinement			Quotient computation		
Model	States	Blocks	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
cycling-4	431101	282943	26.72	2.60	10.29×	59.51	3.32	17.90×
cycling-5	2326666	1424914	170.28	19.42	8.77×	294.15	13.48	21.83×
fgf	80616	38639	8.86	0.88	10.04×	7.42	0.73	10.20×
p2p-5-6	2^{30}	336	26.96	2.99	9.03×	10.25	1.41	7.29×
p2p-6-5	2^{30}	266	9.49	1.21	7.82×	3.67	0.55	6.71×
p2p-7-5	2^{35}	336	24.01	2.97	8.08×	9.26	1.19	7.79×
polling-16	1572864	98304	118.50	10.18	11.64×	66.25	4.49	14.75×
polling-17	3342336	196608	303.65	22.58	13.45×	161.74	10.02	16.14×
polling-18	7077888	393216	705.22	49.81	14.16×	359.49	21.68	16.58×
LTSMIN LTS models			Signature refinement			Quotient computation		
Model	States	Blocks	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
brp-3-4-4	40,592	10,326	13.50	0.92	14.75×	2.45	0.14	17.53×
brp-4-4-4	109,422	27,106	38.91	2.23	17.43×	9.84	0.52	18.93×
franklin-3-3	41,401	883	24	1.24	19.40×	3.13	0.19	16.46×
franklin-4-2	272,241	10,706	330.56	14.67	22.53×	28.04	1.43	19.63×
hesselink-4	142,081,536	6,036	51.41	3.56	14.44×	7.01	1.21	5.78×
hesselink-5	883,738,000	11,005	179.85	12.61	14.26×	22.32	3.64	6.14×
swp-2-4	2,589,056	69,555	267.46	11.33	23.60×	30.78	1.39	22.21×
swp-3-3	1,652,724	65,025	142.60	6.13	23.26×	24.89	1.11	22.39×
swp-4-3	7,429,632	264,708	630.73	25.92	24.34×	111.69	4.55	24.56×

Table 4 Computation time in seconds for partition refinement and quotient computation on various benchmarks provided with the original SIGREF tool and generated by LTSMIN

of quantitative properties, e.g., performance and dependability. We implemented strong bisimulation and branching bisimulation in the SIGREFMC tool. Strong bisimulation preserves both internal behavior (τ-transitions) and observable behavior, while branching bisimulation abstracts from internal behavior. The SIGREFMC tool also connects to the LTSMIN tool described in Section 6.2, enabling the minimization of models described with various input languages.

Bisimulation minimization is performed in two steps. The first step is computing the maximal bisimulation, which is a partition computed using signature refinement. The second step is computing the quotient of the original model and the partition, resulting in the minimised system.

See Table 4 for a selection of the benchmark results from [67]. We used benchmark models provided with the original SIGREF tool by Wimmer et al. [70] and from process algebra (in the MCRL2 language) prepared using LTSMIN. See further [67] for a description of the models. The benchmarks were performed on a 48-core machine, consisting of four AMD Opteron$^{\text{TM}}$ 6168 processors with 12 cores each and 128 GB of internal memory. Table 4 shows that the parallel speedup varies with the model used, similarly to the results we obtained with symbolic reachability in Table 3. We obtained speedups of up to 24$\times$ for both the signature refinement step and the quotient computation step.

6.4 *Probabilistic Model Checking*

Sylvan has also been used as a symbolic back-end in the model checker ISCASMC, a probabilistic model checker [30] written in Java. A recent study [60] compared the performance of the BDD libraries CUDD, BuDDy, CacBDD, JDD, Sylvan and BeeDeeDee when used as the symbolic back-end of ISCASMC and performing symbolic reachability.

They summarize the overall runtimes by the following table [60]:

back-end	time (s)	back-end	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 variant 2 of the nodes table in Sylvan. As the results show, Sylvan is competitive with other BDD implementations when used sequentially (with one worker) and benefits from parallelism (with seven workers).

7 Conclusions

This chapter has discussed the two basic ingredients to achieve scalable binary decision diagrams in a multi-core shared-memory environment. The first ingredient is a fine-grained work-stealing framework that provides parallel execution and load balancing of the decision diagram operations. The second ingredient consists of the concurrent, lock-free hash tables for the unique table and the operation cache.

We discussed Sylvan, a parallel implementation of decision diagrams. Sylvan offers an easy-to-use, sequential interface like a traditional BDD package, but with a parallel implementation of its operations. Thus, existing sequential algorithms that depend on decision diagram operations benefit from the multi-core parallelization offered by Sylvan. In addition, sequential algorithms can further profit from the parallel work-stealing framework embedded in Sylvan by implementing parallel tasks that call decision diagram operations in parallel. For example, a transition system can be partitioned and the partitioned transition relations can be applied in parallel via tree-like reductions.

The approach presented in this chapter is versatile. As shown with the different types of decision diagrams implemented by Sylvan and used in the specific applications, the principles of parallel decision diagram operations can be applied to BDDs, MTBDDs, list decision diagrams, multi-way decision diagrams, zero-suppressed decision diagrams, etc. As decision diagrams are heavily used in many application domains, we foresee that parallel decision diagram operations can be a practical tool to bring parallelization to these domains. Future directions also include tackling the challenges that other applications bring, such as efficient dynamic variable reordering and tentative execution of decision diagram operations. Furthermore, the development of parallel decision diagram operations for heterogeneous systems such as clusters of multi-core computers [47, 48] and systems with many cores and highly specialized hierarchies such as GPUs [68] offers additional challenges for BDD operations that need to be addressed in the future.

References

1. Magdy S. Abadir and Hassan K. Reghbati. Functional Test Generation for Digital Circuits Described Using Binary Decision Diagrams. *IEEE Trans. Computers*, 35(4):375–379, 1986.
2. Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPOPP*, pages 219–228. ACM, 2013.
3. S.B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, C-27(6):509–516, 6 1978.
4. Prakash Arunachalam, Craig M. Chase, and Dinos Moundanos. Distributed binary decision diagrams for verification of large circuit. In *ICCD*, pages 365–370, 1996.
5. R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *ICCAD 1993*, pages 188–191, 1993.
6. Debashis Bhattacharya, Prathima Agrawal, and Vishwani D. Agrawal. Test Generation for Path Delay Faults Using Binary Decision Diagrams. *IEEE Trans. Computers*, 44(3):434–447, 1995.
7. F. Bianchi, Fulvio Corno, Maurizio Rebaudengo, Matteo Sonza Reorda, and Roberto Ansaloni. Boolean function manipulation on a parallel system using BDDs. In *HPCN Europe*, pages 916–928, 1997.
8. Stefan Blom and Simona Orzan. Distributed Branching Bisimulation Reduction of State Spaces. *ENTCS*, 89(1):99–113, 2003.
9. Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and Symbolic Reachability. In *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.
10. Robert D. Blumofe. Scheduling multithreaded computations by work stealing. In *FOCS*, pages 356–368. IEEE Computer Society, 1994.
11. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.
12. Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA 2007*, volume 4762 of *LNCS*, pages 162–176. Springer, 2007.
13. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *DAC*, pages 40–45, 1990.

14. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, C-35(8):677–691, 8 1986.
15. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
16. J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 4 1994.
17. G.P. Cabodi, S. Gai, and M. Sonza Reorda. Boolean function manipulation on massively parallel computers. In *Proc. of 4th Symp. on Frontiers of Massively Parallel Computation*, pages 508–509. IEEE, 10 1992.
18. Jer-Sheng Chen and P. Banerjee. Parallel construction algorithms for BDDs. In *ISCAS 1999*, pages 318–322. IEEE, 1999.
19. Ming-Ying Chung and Gianfranco Ciardo. Saturation NOW. In *QEST*, pages 272–281. IEEE Computer Society, 2004.
20. Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In *TACAS*, volume 2031 of *LNCS*, pages 328–342, 2001.
21. Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? In *PDMC*, pages 1–17, 2009.
22. Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In *DAC*, pages 54–60, 1993.
23. Jonathan Ezekiel, Gerald Lüttgen, and Gianfranco Ciardo. Parallelising symbolic state-space generators. In *CAV*, volume 4590 of *LNCS*, pages 268–280, 2007.
24. Karl-Filip Faxén. Wool-A work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100, 2008.
25. Karl-Filip Faxén. Efficient work stealing for fine grained parallelism. In *ICPP 2010*, pages 313–322. IEEE Computer Society, 2010.
26. Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE 2005*, pages 196–205. ACM, 2005.
27. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223. ACM, 1998.
28. S. Gai, M. Rebaudengo, and M. Sonza Reorda. An improved data parallel algorithm for Boolean function manipulation using BDDs. In *Proc. Euromicro Workshop on Par. and Distrib. Processing*, pages 33–39. IEEE, 1 1995.
29. Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.
30. Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. iscasmc: A web-based probabilistic model checker. In *FM*, volume 8442 of *LNCS*, pages 312–317. Springer, 2014.
31. Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer Berlin / Heidelberg, 2000.
32. Masakazu Ishihata, Taisuke Sato, and Shin-ichi Minato. Compiling bayesian networks for parameter learning based on shared bdds. In *AI 2011*, volume 7106 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2011.
33. Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS 2015*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.
34. S. Kimura and E.M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. of IC on Computer Design: VLSI in Computers and Processors ICCD*, pages 220–223, 9 1990.

35. S. Kimura, T. Igaki, and H. Haneda. Parallel Binary Decision Diagram Manipulation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, E75-A(10):1255–62, 10 1992.
36. Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting multi-core reachability performance with shared hash tables. In *FMCAD 2010*, pages 247–255. IEEE, 2010.
37. Alfons W. Laarman, Jaco van de Pol, and Michael Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *LNCS*, pages 506–511. Springer, 2011.
38. Elsa Loekito and James Bailey. Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams. In *SIGKDD 2006*, pages 307–316. ACM, 2006.
39. Alberto Lovato, Damiano Macedonio, and Fausto Spoto. A Thread-Safe Library for Binary Decision Diagrams. In *SEFM*, volume 8702 of *LNCS*, pages 35–49. Springer, 2014.
40. Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD 1998*, pages 6–9, 1988.
41. Yusuke Matsunaga and Masahiro Fujita. Multi-level logic optimization using binary decision diagrams. In *ICCAD 1989*, pages 556–559. IEEE, 1989.
42. Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. Read, Write and Copy Dependencies for Symbolic Model Checking. In Eran Yahav, editor, *HVC*, volume 8855 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2014.
43. Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A parallel BDD package. In *FMCAD*, pages 501–507, 1998.
44. Shin-ichi Minato. Techniques of BDD/ZDD: Brief History and Recent Activity. *IEICE Transactions*, 96-D(7):1419–1429, 2013.
45. Shin-ichi Minato, Ken Satoh, and Taisuke Sato. Compiling bayesian networks by symbolic probability calculation based on zero-suppressed bdds. In *IJCAI 2007*, pages 2550–2555, 2007.
46. Hiroyuki Ochi, Nagisa Ishiura, and Shuzo Yajima. Breadth-first manipulation of SBDD of Boolean functions for vector processing. In *DAC*, pages 413–416, 1991.
47. Wytse Oortwijn. Distributed Symbolic Reachability Analysis. Master's thesis, University of Twente, Dept. of C.S., 2015.
48. Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. Distributed Binary Decision Diagrams for Symbolic Reachability. In *SPIN*, 2017.
49. Jörn Ossowski. *JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn, 10 2010.
50. Yegnashankar Parasuram, Edward P. Stabler, and Shiu-Kai Chin. Parallel implementation of BDD algorithms using a distributed shared memory. In *HICSS (1)*, pages 16–25, 1994.
51. Radek Pelánek. BEEM: benchmarks for explicit model checkers. In *SPIN*, pages 263–267, Berlin, Heidelberg, 2007. Springer-Verlag.
52. Karen A. Reay and John D. Andrews. A fault tree analysis strategy using binary decision diagrams. *Rel. Eng. & Sys. Safety*, 78(1):45–56, 2002.
53. Yuko Sakurai, Suguru Ueda, Atsushi Iwasaki, Shin-ichi Minato, and Makoto Yokoo. A compact representation scheme of coalitional games based on multi-terminal zero-suppressed binary decision diagrams. In *PRIMA 2011*, volume 7047 of *Lecture Notes in Computer Science*, pages 4–18. Springer, 2011.
54. Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierchachy. In *DAC*, pages 635–640, 1996.
55. Mathias Soeken, Laura Tague, Gerhard W. Dueck, and Rolf Drechsler. Ancilla-free synthesis of large reversible functions using binary decision diagrams. *J. Symb. Comput.*, 73:1–26, 2016.
56. Fabio Somenzi. Efficient manipulation of decision diagrams. *STTT*, 3(2):171–181, 2001.
57. Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2015.

58. Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *DAC*, pages 641–644, 1996.
59. Tom van Dijk. *Sylvan: Multi-core Decision Diagrams*. PhD thesis, University of Twente, 7 2016.
60. Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking. In *SETTA*, volume 9409 of *LNCS*, pages 35–51. Springer, 2015.
61. Tom van Dijk, Alfons Laarman, and Jaco van de Pol. Multi-core BDD operations for symbolic reachability. *ENTCS*, 296:127–143, 2013.
62. Tom van Dijk, Alfons W. Laarman, and Jaco van de Pol. Multi-core and/or Symbolic Model Checking. *ECEASST*, 53, 2012.
63. Tom van Dijk and Jaco van de Pol. Lace: Non-blocking Split Deque for Work-Stealing. In *MuCoCoS*, volume 8806 of *LNCS*, pages 206–217. Springer, 2014.
64. Tom van Dijk and Jaco van de Pol. Sylvan: Multi-Core Decision Diagrams. In *TACAS*, volume 9035 of *LNCS*, pages 677–691. Springer, 2015.
65. Tom van Dijk and Jaco van de Pol. Multi-Core Symbolic Bisimulation Minimisation. In *TACAS*, volume 9636 of *LNCS*, pages 332–348. Springer, 2016.
66. Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2016.
67. Tom van Dijk and Jaco van de Pol. Multi-core symbolic bisimulation minimisation. *International Journal on Software Tools for Technology Transfer*, 2017. Submitted.
68. Miroslav N. Velev and Ping Gao. Efficient parallel GPU algorithms for BDD manipulation. In *ASP-DAC*, pages 750–755. IEEE, 2014.
69. David B. Wagner and Brad Calder. Leapfrogging: A portable technique for implementing efficient futures. In *PPOPP*, pages 208–217. ACM, 1993.
70. Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref-A Symbolic Bisimulation Tool Box. In *ATVA*, volume 4218 of *LNCS*, pages 477–492. Springer, 2006.
71. Bwolen Yang and David R. O'Hallaron. Parallel breadth-first BDD construction. In *PPOPP*, pages 145–156, 1997.