Distributed Binary Decision Diagrams for Symbolic Reachability

Wytse Oortwijn
University of Twente
P.O.-box 217, 7500 AE
Enschede, the Netherlands
w.h.m.oortwijn@utwente.nl

Tom van Dijk
Johannes Kepler University
Altenbergerstr. 69
Linz, Austria
tom.vandijk@jku.at

Jaco van de Pol
University of Twente
P.O.-box 217, 7500 AE
Enschede, the Netherlands
j.c.vandepol@utwente.nl

ABSTRACT

Decision diagrams are used in symbolic verification to concisely represent state spaces. A crucial symbolic verification algorithm is reachability: systematically exploring all reachable system states. Although both parallel and distributed reachability algorithms exist, a combined solution is relatively unexplored. This paper contributes BDD-based reachability algorithms targeting compute clusters: high-performance networks of multi-core machines. The proposed algorithms may use the entire memory of every machine, allowing larger models to be processed while increasing performance by using all available computational power. To do this effectively, a distributed hash table, cluster-based work stealing algorithms, and several caching structures have been designed that all utilise the newest networking technology. The approach is evaluated extensively on a large collection of models, thereby demonstrating speedups up to 51.1x with 32 machines. The proposed algorithms not only benefit from the large amounts of available memory on compute clusters, but also from all available computational resources.

KEYWORDS

Symbolic model checking, parallel computing, distributed computing, high-performance computing, reachability analysis

ACM Reference format:
Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. 2017. Distributed Binary Decision Diagrams for Symbolic Reachability. In *Proceedings of the 24th International SPIN Symposium on Model Checking of Software, Santa Barbara, California, United States, July 2017 (SPIN'17),* 10 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Reachability analysis is a crucial component in model checking as it allows verifying temporal safety properties. Reachability analysis systematically explores all reachable states of a given system. It is computationally demanding, since its time (and for many approaches space) requirements increase exponentially with the number of processes and variables in the input model. This phenomenon is widely known as the state space explosion problem.

In many cases the space requirements of reachability analysis can be significantly lowered [10] by representing the set of visited states symbolically [33], for example as Binary Decision Diagrams (BDDs). Another approach to lower space and time requirements is to exploit parallel computing power. In a distributed setting the memory of several machines can be combined, allowing larger models to be processed, while in multi-core approaches speedup can be obtained due to parallelism. However, BDD manipulations are hard to parallelise as they are very memory intensive and their memory access patterns are irregular [16].

This paper contributes new algorithms for BDD-based reachability that combine these approaches, targeting clusters of multi-core machines connected via a high-speed network. The novelty of the approach is that we use the design of a shared-memory, multi-core BDD package, viz. Sylvan [39], and project it onto *distributed* networks of *multi-core* machines by using the newest networking technology. In particular, Infiniband networks [1] with support for RDMA (Remote Direct Memory Access) are targeted, which allow to minimise the cost and overhead of network communication.

This setting requires a careful redesign of the basic ingredients of a scalable BDD package, namely: a distributed concurrent hash table that represents the BDD unique table and the operation cache; and a distributed fine-grained load balancer that schedules the parallel invocations of the recursive BDD operations. The main challenge in the design of these components is to minimise the amount and overhead of generated networking operations. We propose an efficient lockless distributed hash table with operations that exploit RDMA. For load balancing we propose lockless work stealing algorithms that take the processing hierarchy into account. Existing work on distributed hash tables [15, 29, 36] and load balancers [13, 14, 27] either rely on locking or generate more network traffic, which have negative effects on both performance and scalability.

The BDD operations have been experimentally evaluated by calculating the sets of reachable states of a large collection of models, including well-known BEEM models and Petri nets, on a high-performance compute cluster. Compared to sequential runs, speedups up to 51.1x on 36 16-core machines are obtained, which demonstrates a significant improvement over existing work. To our knowledge, we report the first distributed/multi-core BDD package. Moreover, although this paper focusses on BDDs the implementation may very well be adjusted to support other decision diagram variants, including MDDs, LDDs, and MTBDDs, as in [34, 39].

Related work on parallel BDD packages include BuDDy [12] and CUDD [34]. Other BDD libraries like BeeDeeDee [24] and Sylvan [39] apply parallelism on multi-core, shared memory machines, where each worker can access every BDD node. Good performance and scalability are reported. These libraries are all optimised for NUMA architectures in a single-machine setting.

An earlier line of research, including [8, 9, 16, 19, 26], targeted high-speed networks of workstations instead to exploit their combined memory. Their algorithms are based on message passing.

However, their time efficiency is limited, since network communication easily becomes a performance bottleneck [10]. BDDNOW [26] was the first distributed BDD package claiming some speedup on a network of workstations. The BDDs can be distributed in two ways; Grumberg et al. [19] apply vertical slicing [20] to partition the work among machines, and Chung and Ciardo propose horizontal slicing for the saturation strategy [8]. Effectively, slicing limits the amount of available parallel work, but the speedup was improved by performing speculative image computations [9]. Other work targets distributed shared memory (DSM) architectures [7, 31] to implement BDD algorithms using a standard depth-first approach.

The paper is organised as follows. Section 3 discusses the design of the distributed BDD table. In Section 4 the operations for hierarchical work stealing are presented. Section 5 shows the implementation of the BDD algorithms by combining work stealing with the concurrent hash table. Experimental results are presented in Section 6. Finally, Section 7 summarises our conclusion.

2 PRELIMINARIES

This section recalls the basic BFS algorithm for symbolic reachability. The algorithm is defined using an *if-then-else* operation on BDDs and an operation for calculating the *relational product*, which we also define. Section 5 presents distributed implementations of these two BDD operations. Furthermore, this section discusses the shared memory model that is used by our implementation.

2.1 Symbolic Reachability

The following definition is based on the work of [6, 39].

Definition 2.1. A *(reduced ordered) BDD* is a directed acyclic graph with the following properties:

(1) There is a single root node and two terminal nodes 0 and 1.
(2) Each non-terminal node p has a variable $var(p) = x_i$ and two outgoing edges, labelled "*low*" and "*high*". We use the notations $lvl(p) = i$, $p.low = q_0$, and $p.high = q_1$.
(3) For each edge from node p to non-terminal node q we have $lvl(p) < lvl(q)$ (i.e. the BDD is ordered).
(4) There are no redundant nodes, that is, nodes p with $p.low = p.high$ (i.e. the BDD is reduced).
(5) There are no duplicate nodes, that is, nodes p, q with $p \neq q$ and $lvl(p) = lvl(q) \wedge p.low = q.low \wedge p.high = q.high$.

BDDs are used to efficiently represent Boolean functions [6]. Most logical operators, including conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$) can be performed on the BDD representations of their operands via the ITE operator.

Definition 2.2 (If-then-else). Let $\phi, \psi, \rho : \mathbb{B}^n \rightarrow \mathbb{B}$ be three Boolean functions and A, B, C their respective BDD representatives. The *if-then-else operator*, denoted by $\texttt{ITE}(A, B, C)$, is defined as the BDD representing the function $(\phi \wedge \psi) \vee (\neg\phi \wedge \rho)$.

Definition 2.3 (Transition Systems). A *transition system* is a triple $(S, S_I, \rightarrow)$, with S a set of states, $S_I \subseteq S$ a set of initial states, and $\rightarrow \subseteq S \times S$ a transition relation.

Let $\mathbb{B}^n$ be the set of all n-sized Boolean vectors and $T = (\mathbb{B}^n, S_I, \rightarrow)$ be a (Boolean) transition system. Then T can be represented by Boolean membership functions; the initial states can be represented

Algorithm 1: Symbolic reachability

```
def Reach($I, T, X, X'$):
    $States \leftarrow I$
    $Prev \leftarrow 0$
    while $States \neq Prev$:
        $Succ \leftarrow$ RelProd($States, T, X, X'$)
        $Succ \leftarrow$ Rename($X', X, Succ$)
        $Prev \leftarrow States$
        $States \leftarrow$ ITE($States, 1, Succ$)
    return $States$
```

by a function $\phi : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $S_I = \{s \in \mathbb{B}^n \mid \phi(s)\}$ and the transition relation can be represented by a function $\psi : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$ such that $\forall x, y \in \mathbb{B}^n.\ \psi(x, y) \Leftrightarrow (x, y) \in \rightarrow$. The problem of finding the set of reachable states of T can now be reduced to finding a fixed-point of the following series:

$$\phi_0(s) \equiv \phi(s)$$
$$\phi_{i+1}(s) \equiv \phi_i(s) \vee \exists s'.(\phi_i(s') \wedge \psi(s', s))$$

To clarify, the ψ-successors of $\phi_i(s)$ are obtained via $\exists s'.(\phi_i(s') \wedge \psi(s', s))$ and reachability is performed by repeatedly finding ψ-successors, starting from ϕ, until a fixed point is reached, that is, until $\phi_j(s) = \phi_{j+1}(s)$ for some $j \geq 0$. By computing reachability over the BDD representations of ϕ and ψ, *symbolic reachability* over the transition system T is performed. Finding ψ-successors with BDDs involves calculating relational products.

Definition 2.4 (Relational Product). Let $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ be two sets of (disjoint) variables, $\phi : \mathbb{B}^n \rightarrow \mathbb{B}$ a Boolean function, and $\psi : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$ a Boolean relation. Let A and B be the respective BDD representations of the functions ϕ and ψ. The *relational product* over A and B with respect to X', denoted by $\texttt{RelProd}(A, B, X, X')$, is the BDD representing the function $\exists X.(\phi(X) \wedge \psi(X, X'))$.

Algorithm 1 presents a basic BFS symbolic reachability algorithm, named $\texttt{Reach}(I, T, X, X')$, where X and X' are two (disjoint) sets of variables and I and T the BDDs representing the initial state and the transition relation, respectively. The algorithm uses RelProd to find successors of *States* and uses ITE to unify *States* and *Succ* via disjunction (by calculating $States \vee Succ$). The Rename operation simply renames all occurrences of variables in X' by the matching variables in X in the BDD *Succ*. Reach terminates when a fixed point has been found, which happens when $States = Prev$.

2.2 Shared Memory Abstraction

The algorithms proposed in this paper target Infiniband networks with support for Remote Direct Memory Access (RDMA). In particular, *one-sided* RDMA operations are used, which are network operations that access the memory of a remote machine without invoking its CPU. One-sided RDMA is highly optimised to minimise overhead from the OS kernel and relieves the remote CPU from handling incoming network traffic. As a result, compared to TCP over Ethernet, Infiniband lowers latency by an order of magnitude when using one-sided RDMA [29]. This motivated a renewed attempt to directly distribute BDD operations.

On top of this network model sits a Partitioned Global Address Space (PGAS) [11] software layer that provides the required shared-memory abstraction to our BDD algorithms. PGAS combines the shared and distributed memory models and thereby exposes data-locality. The PGAS abstraction allows to allocate memory uniformly over a network of machines and provides a shared-memory interface to this memory. All machine-local memory accesses are then handled by the local CPU and all remote memory accesses are translated to one-sided RDMA operations.

We now introduce some notation. In a PGAS environment with n threads, named $t_0, \dots, t_{n-1}$, a shared array B can be uniformly allocated if B can be indexed $\mathsf{B}[0], \dots, \mathsf{B}[kn-1]$ for some $k > 0$. The PGAS abstraction distributes B equally over the n threads, so that each thread owns k consecutive entries of B. For any thread t_i we write $t_i.\mathsf{B}[0], \dots, t_i.\mathsf{B}[k-1]$, or simply $t_i.\mathsf{B}$ to denote the k entries of B owned by t_i. To distribute B, the PGAS abstraction allocates memory on each machine according to the thread layout, so that $t_i.\mathsf{B}$ is *local memory* for the thread t_i. As an advantage, threads may benefit from exploiting data-locality and the block symmetry makes programming easier. Standard reads and writes on shared memory, written "$val \leftarrow \mathsf{B}[j]$" and "$\mathsf{B}[j] \leftarrow val$" for some index j, are handled locally by the CPU if $\mathsf{B}[j]$ is local, or via one-sided RDMA in case $\mathsf{B}[j]$ is in remote memory. An atomic $\mathsf{cas}(\mathsf{B}[j], c, v)$ operation is used to prevent races, which takes a shared memory location $\mathsf{B}[j]$ and writes v to $\mathsf{B}[j]$ only if $\mathsf{B}[j]$ contains c. The value at location $\mathsf{B}[i]$ *prior* to calling cas is returned.

We use PGAS in a distributed setting, but heterogeneous and NUMA-aware architectures are also supported. All threads compute in a Single Program Multiple Data (SPMD) fashion.

3 DISTRIBUTED UNIQUE TABLE

The two central data structures of modern BDD packages are the *unique table* and the *operation cache* (sometimes called the *computed table*) [37]. The unique table contains all BDD nodes and is used to avoid duplication. The operation cache stores results of previous BDD operations and is used to avoid repetitive work, which is key to efficient BDD manipulation. Both these tables are implemented as hash tables. Since BDD operations are very memory intensive, a scalable hash table design is crucial for a scalable BDD package.

The major challenge for a *distributed* BDD package is distributing the unique table and operation cache over the network so that their entries can be accessed with minimal access times by any worker needing them. Our approach is to distribute the hash tables by using the PGAS abstraction. One-sided RDMA is exploited to minimise the overhead of finds and inserts. Moreover, an *adaptive* hashing strategy is used to dynamically adjust and reduce the required number of RDMA operations, based on the load-factor. This improves hash table access-times and increases the scalability.

Related work includes Pilaf [29], Nessie [36], and FaRM [15]. Pilaf is a key-value store and assumes a setting with a single server and multiple clients. Lookup operations are handled by clients via one-sided RDMA. However, to avoid read-write races all inserts are handled by the server. The Cuckoo hashing strategy is used to keep the amount of network traffic low. Nessie is a distributed hash table

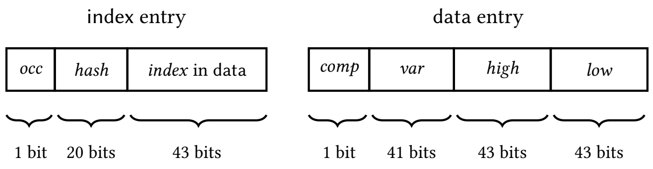


Figure 1: The memory layout of our lockless hash table, implemented as two separate shared arrays, named data **and** index**. The** data **array stores BDD nodes as 128-bit entries and the** index **array is used as an indexing array for** data**.**

that improves on Pilaf by allowing clients to handle both lookups and inserts via RDMA. FaRM implements a hash table that uses a variant of Hopscotch hashing that is more efficient than Cuckoo hashing in terms of required network operations. FaRM aims to minimise latency, but at the cost of CPU activity.

Our approach improves by using linear probing instead of Cuckoo hashing or Hopscotch hashing [30]. Since linear probing examines buckets that are consecutive in memory, a *range* of buckets can be obtained with a single RDMA operation and then examined in local memory. As the load-factor increases, our hashing strategy also increases the range of buckets obtained by the RDMA operation. This approach reduces network traffic when the load-factor increases and thus contributes to the scalability. In contrast, Cuckoo hashing already requires multiple RDMA reads per probe and Hopscotch hashing becomes less efficient when the load-factor increases [30].

The remaining section discusses the unique table, in particular its memory layout and the operations for finding and inserting BDD nodes. The operation cache can be implemented as a simplified version of the unique table and is discussed in Section 5.

3.1 Memory Layout

Figure 1 shows the memory layout of the unique table, which is implemented as two separate arrays, named index and data. The data array contains BDD nodes and index is used as an indexing array for data. Both arrays are uniformly distributed over all machines in the network via the shared-memory PGAS abstraction.

Using two separate arrays has multiple advantages for BDD packages [35, 40]. Firstly, this design allows garbage collection to be implemented efficiently, as reorganising index can be done without affecting the entries of data. Secondly, this design allows to save on RDMA operations by restricting hash table insertions to only write to *local memory* regions of data. To make this work, data maintains the invariant of a global-read, local-write array; each thread t_i may read any part of data but is only permitted to write in its *local* block t_i.data. The index array is then used as a global indexing array; in contrast to data, each thread may atomically write to every entry in index via compare-and-swap.

Figure 1 shows that every BDD node B stored in data is stored as an 128-bit entry and consists of four components. The $B.var$ component holds the BDD variable and the components $B.high$ and $B.low$ represent the high edge and low edge of B, respectively. The high- and low edges are simply addresses to other entries in data. The last component, $B.comp$, holds a single bit for representing *complement edges* [28]. If $B.comp$ is set, B is negated by switching

Algorithm 2: Lookups and Inserts (on thread t_i)

```
1  def $t_i$.LookupOrInsert($B$):
2      data[$t_i$.localpos] $\leftarrow B$
3      $rangesize \leftarrow$ CurrentRange()
4      for $j \leftarrow 0$ to threshold:
5          $range \leftarrow$ FetchRange($j, rangesize$)
6          for $k \leftarrow 0$ to $rangesize - 1$:
7              if $range[k].occ$:
8                  $index \leftarrow$ CheckNode($range[k], B$)
9                  if $index \neq$ invalid: return $index$
10             else
11                 $b \leftarrow$ MakeNewEntry($t_i.localpos, B$)
12                 $addr \leftarrow$ AddressOf($range[k]$)
13                 $b' \leftarrow$ cas(index[$addr$], $range[k], b$)
14                 if $b' = range[k]$:
15                     $t_i.localpos \leftarrow t_i.localpos + 1$
16                     return $t_i.localpos - 1$
17                 $index \leftarrow$ CheckNode($b', B$)
18                 if $index \neq$ invalid: return $index$
19     return full
```

Algorithm 3: Retrieving BDD nodes from data

```
1  def FetchNode($b$):
2      $B \leftarrow$ FindInCache($b$)
3      if $B =$ notfound:
4          $B \leftarrow$ data[$b.index$]
5          PutInCache($b, B$)
6      return $B$

7  def CheckNode($b, B$):
8      $hash \leftarrow$ hashvalue($B$)
9      if $b.hash$ matches $hash$:
10         if $B =$ FetchNode($b$):
11             return $b.index$
12     return invalid
```

its terminals, thereby allowing its subgraphs to be reused. As a result, BDD negations become trivial to implement.

Each entry b in index consists of three components. The first is $b.occ$, which is a single bit that denotes whether the entry b is occupied (currently in use). The third component is $b.index$, which simply holds an address to an entry in data (that is, to a BDD node). Assuming that data$[b.index] = B$, the second component $b.hash$ is a *fragment* of the hash value of B. This fragment is stored to sometimes prevent reading from data during hash table operations. For example, by inspecting $b.hash$ it may already be clear that data$[b.index]$ does not contain the intended BDD node. In such cases, consulting $b.hash$ prevents an unneeded RDMA read.

3.2 Lookups and Inserts

Our hash table supports a single operation, named LookupOrInsert, that takes a BDD node B as argument and inserts B into data if B has not been inserted before. Testing whether data already contains B is done efficiently by consulting the index table. LookupOrInsert returns either the index of B in data, or the value "full" indicating that B has not been found nor been inserted.

Algorithm 2 shows the implementation of LookupOrInsert on thread t_i. Recall that threads only write to parts of data that are local to them. To maintain this invariant, LookupOrInsert starts by writing B into data$[t_i.localpos]$ on line 2, where $t_i.localpos$ is t_i's index to the next free entry in the local block t_i.data.

After that, the *range size* is determined on line 3 by invoking CurrentRange. The range size determines the number of entries obtained from index to be examined via linear probing. CurrentRange increases the sizes of ranges when the load-factor increases. The j-th consecutive range is obtained from index via the FetchRange operation on line 5 and is stored in the array *range*. The loop on line 6 then iterates over *range* in the fashion of linear probing.

When considering the k-th iteration over *range*, if $range[k]$ is occupied we check if B matches with data$[range[k].index]$, thereby determining whether B has already been inserted. This check is done via CheckNode on line 8, whose implementation is given in Algorithm 3. Note that CheckNode only consults data when the hash value of B matches with the hash fragment stored in $range[k]$. Moreover, an extra caching layer is added internally to further reduce the number of generated RDMA calls (see FetchNode).

If, however, $range[k]$ is unoccupied the algorithm tries to claim the corresponding entry in index on line 13. To prepare for this a new entry for index is created (line 11) and the address in index corresponding to $range[k]$ is calculated (line 12). If the claim succeeds, $t_i.localpos$ is incremented and returned on line 16, otherwise some other worker has claimed index$[addr]$ in between lines 5 and 13. It may happen that the other worker inserted B at that location, therefore the check at line 18 is needed to ensure correctness.

Instead of linear probing we considered using quadratic probing for retrieving consecutive ranges (on line 5), as suggested by Laarman et al. [23]. Quadratic probing reduces clustering, which is a known problem of linear probing. However, to preserve correctness when using quadratic probing the range sizes (line 3) need to be static. We implemented both strategies but found linear probing to be more efficient in terms of generated network traffic due to the dynamic range sizes. We did not observe serious clustering issues. Nonetheless, both strategies are implemented, so if the user finds himself limited by clustering he can switch strategy.

4 HIERARCHICAL WORK STEALING

BDD operations are generally defined recursively. Parallel BDD packages like Sylvan apply fine-grained task parallelism by encapsulating each recursive call as a task. Every thread maintains a local task pool on which these tasks can be pushed or popped. When a thread becomes *idle* (runs out of tasks) it attempts to steal work from remote task pools. In this scenario the stealing thread is referred to as the *thief* and the targeted thread as the *victim*.

Many work stealing frameworks [17, 38], including frameworks designed for compute clusters [13, 14, 27] implement their task pools as *split deques*. Split deques are double-ended queues that are partitioned into a *public* and a *private* region via a dynamic split point. Thieves may only steal from the public regions of remote deques and steal operations are handled without the victim's help. Deque owners may adjust their split point to adapt the public region. However, the operations for task stealing and split point adaption

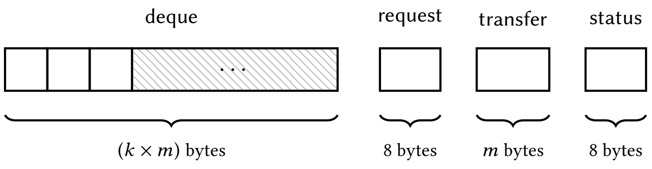


Figure 2: Per-thread memory organisation for private-deque work stealing, with k the number of entries in deque **and m the size in bytes of each task.**

often rely on locking for their correctness, which has negative effects on scalability, especially in a distributed setting.

For our distributed setting we designed algorithms for lockless *private-deque* work stealing. Private deques do not have public regions and are maintained entirely in private memory. Steal operations are performed by sending the victim a *steal request* and the victim responds by sending the thief part of the tasks in its deque. Compared to using split deques this approach is lockless, but requires active participation of victims. However, this is compensated for by handling steal requests while waiting for pending RDMA operations, thereby making effective use of the victim's idle-times.

Related work includes Wool [17] and Lace [38], which are light-weight variants of Cilk [5], all frameworks for fine-grained task parallelism. The multi-core BDD package Sylvan [39] demonstrates good parallel scalability in symbolic reachability by using Lace. None of these frameworks, however, considers the processing hierarchy. The cluster-based task-parallel frameworks HotSLAW [27] and Scioto [13, 14] target networks of multi-core machines and exploit the locality hierarchy. However, both frameworks rely on locking for their correctness, which severely impacts performance.

Figure 2 presents the per-thread memory organisation of our setting. Each thread t_i maintains a task pool t_i.deque, handles steal requests and responses via t_i.request and t_i.transfer, and uses t_i.status for termination detection. Moreover, status can be used to initiate a stop-the-world scenario, for example needed by garbage collection. Each memory component is shared, including deque, since threads are allowed to write back the results of stolen tasks directly into remote deques via one-sided RDMA.

4.1 Task Parallelism

Like most other frameworks for task parallelism our implementation uses the operations **spawn**, **call**, and **sync** to spawn and execute new tasks, and synchronise on tasks, respectively. The notation t_i.**spawn** is used to indicate that the **spawn** is performed by thread t_i (and likewise for t_i.**call** and t_i.**sync**). More specifically, t_i.**spawn**(T) pushes a task T onto t_i.deque.*tail*, thereby allowing other threads to steal T while t_i is working on other tasks. The t_i.**call**(T) operation executes a task T on the calling thread t_i without adding it to t_i.deque beforehand. Finally, t_i.**sync**() pops a task T from t_i.deque.*tail* and executes it, or returns T's result in case T was stolen and the result has been written back by the thief.

To illustrate the use of task parallelism, Algorithm 4 presents two (functionally equivalent) algorithms for calculating Fibonacci: a standard sequential, recursive version, `FibSeq`; and a task-parallel version, `FibPar`, in which the recursive calls are translated to tasks.

Algorithm 4: Sequential and Parallel Fibonacci

```
def FibSeq($n$):
    if $n < 2$: return $n$
    $a \leftarrow$ FibSeq($n-1$)
    $b \leftarrow$ FibSeq($n-2$)
    return $a + b$

def FibPar($n$):
    if $n < 2$: return $n$
    spawn(FibPar, $n-1$)
    $r \leftarrow$ call(FibPar, $n-2$)
    return $r$ + sync()
```

Technically, the task-parallel version starts by assigning the initial task "(`FibPar`, n)" to the thread t_0. All other threads eagerly attempt to obtain work via `steal` operations and eventually succeed. Due to the stack-like invocations of **spawn** and **sync**, t_0 is also the last thread to execute a task. When this happens t_0 writes "done" to t_i.status on every thread t_i, thereby indicating that they may also terminate. Finally, every thread t_i terminates when it reads "done" in t_i.status.

4.2 Work Stealing

The stealing procedure is handled internally by two operations. The first is t_i.`steal`(j), called by a thief t_i to attempt a steal from a chosen victim thread t_j. The second operation is t_j.`communicate`() and is used by the victim t_j to handle the steal request. The following three steps are taken when a thread t_i attempts to steal from t_j:

(1) Thread t_i invokes `steal`(j), which atomically writes a "⟨steal, i⟩" message into t_j.request by applying `cas`. If `cas` succeeds, t_i waits until t_j has written a response to t_i.transfer. Waiting is done by continuously polling on t_i.transfer.
(2) Thread t_j invokes `communicate`(), which simply checks if t_j.request contains a "⟨steal, i⟩" message. In that case, if t_j.deque.*head* holds a task T that can be stolen, then t_j writes a "⟨response, T⟩" message into t_i.transfer, otherwise an empty message "⟨response⟩" is written back. To minimise latency and idle-times of both thieves and victims, `communicate` is called as often as possible, mostly while waiting for pending RDMA operations.
(3) Thread t_i receives a "response" message from t_j. If it contains a task T, then T is executed and its result is written to the original task entry in t_j.deque. Otherwise, t_i searches for another victim and tries again.

Successful steals require merely three one-sided RDMA writes (1.5 network round-trips): one for the atomic write to t_j.request, one for writing to t_i.transfer, and one for writing back the result to t_j.deque. Furthermore, `communicate` is non-blocking, meaning that it does not wait for RDMA operations to complete before returning. Therefore, handling steals requires only a very small amount of work and most overhead for stealing resides at `steal`.

4.3 Victim Selection

Before a thief performs a `steal`(j) operation it has to select a victim thread t_j. The victim selection procedure keeps the processing hierarchy into account, meaning that the thief prefers to select a victim that is physically close to it. By doing so, network traffic as well as the idle-times of thieves are further reduced. Hierarchical victim selection has also been shown to achieve better performance compared to a purely randomised victim selection strategy [27].

Our victim selection procedure first uses *leapfrogging* [41], a strategy in which victims steal back from their thieves and thereby obtain part of their original work. An advantage is that leapfrogging limits space requirements of deques to the size required for sequential execution [18]. If leapfrogging fails, victims are randomly selected by considering the processing hierarchy. More specifically, every thread t_i maintains four disjoint sets $t_i.dom_j$ with $j \in \{0, \dots, 3\}$ of thread identifiers that represent the processing hierarchy from t_i's perspective. These sets are constructed so that $t_i.dom_j \subseteq \{0, \dots, n-1\}$ with n the number of active threads, and are defined so that all threads in $t_i.dom_{k+1}$ have higher memory access times than threads in $t_i.dom_k$ relative to t_i. Before considering victims in $t_i.dom_{k+1}$, a thief t_i first attempts $|t_i.dom_k|$ steal attempts on victims randomly selected from $t_i.dom_k$.

Min et al. show that *steal-many* strategies, i.e. stealing more than one task per steal operation may improve scalability [27]. Steal-many strategies may be beneficial when stealing from threads with high memory access times, for example from threads in $t_i.dom_3$. We experimented with several steal-many strategies, including steal-half and hierarchical-adapted stealing [27]. However, in case of BDD operations we observed that skews occur in load-balancing, which negatively impact scalability. We found that a simple steal-1 strategy performs best for BDD operations.

5 DISTRIBUTED REACHABILITY ANALYSIS

This section discusses the unique table and the distributed implementations of the operations needed to perform reachability, ITE and RelProd. Both these operations apply private-deque work stealing and use the distributed hash table discussed earlier. Note that all these components can be reused to implement other BDD operations or to support alternative decision diagram variants.

During distributed symbolic reachability, each thread holds a copy of the fixed (partitioned) transition relation in local memory to save on network operations. The alternative would be having a single shared transition relation, implemented on the PGAS abstraction, but then certain parts of data and thereby certain machines may become hotspots. We implemented both but the local memory variant shows clear performance benefits.

5.1 Operation Cache

The task-parallel implementations of ITE and RelProd rely on dynamic programming to achieve a polynomial time complexity. The intermediate results of their computations are therefore stored in a global *operation cache*. The operation cache is implemented as a lossy hash table similar to the one described in Section 3. Finding and inserting a task T is done via two separate operations: FindInCache(T) and PutInCache(T). The main difference with LookupOrInsert is that hash collisions are not resolved; instead of obtaining a range of entries, only a single entry is considered and collisions are handled simply by overwriting the colliding entry.

Like in Section 3 two separate shared arrays are used to implement the underlying hash table, named cacheindex and cachedata. The cachedata array stores the actual tasks and cacheindex is a global indexing array for cachedata. Furthermore, cachedata is a global-read, local-write array; by executing t_i.PutInCache(T) the thread t_i inserts the given task T *locally* into t_i.cachedata. It would be possible to implement the operation cache as a single shared array, but this would make PutInCache more expensive in terms of network traffic since task sizes may be variable. Currently, PutInCache generates at most one RDMA write. FindInCache generates at most two RDMA reads for successful lookups. Unsuccessful cache lookups require only a single RDMA read.

Algorithm 5: If-then-else (see also Definition 2.2)

```
1  def ITE($b_I, b_T, b_E$):
2      if $b_I = 1$: return $b_T$
3      if $b_I = 0$: return $b_E$
4      $index \leftarrow$ FindInCache(ITE, $b_I, b_T, b_E$)
5      if $index$ = notfound:
6          do in parallel:
7              $B_I \leftarrow$ GetNode($b_I$)
8              $B_T \leftarrow$ GetNode($b_T$)
9              $B_E \leftarrow$ GetNode($b_E$)
10         $x \leftarrow$ TopVariable($B_I, B_T, B_E$)
11         $\langle b_I^0, b_I^1 \rangle \leftarrow \langle$Low($B_I, x$), High($B_I, x$)$\rangle$
12         $\langle b_T^0, b_T^1 \rangle \leftarrow \langle$Low($B_T, x$), High($B_T, x$)$\rangle$
13         $\langle b_E^0, b_E^1 \rangle \leftarrow \langle$Low($B_E, x$), High($B_E, x$)$\rangle$
14         spawn(ITE, $b_I^0, b_T^0, b_E^0$)
15         $high \leftarrow$ call(ITE, $b_I^1, b_T^1, B_E^1$)
16         $low \leftarrow$ sync()
17         $B \leftarrow$ MakeNewNode($x, high, low$)
18         $index \leftarrow$ LookupOrInsert($B$)
19         PutInCache(ITE, $b_I, b_T, b_E, index$)
20     return $index$
```

5.2 If-then-else

Algorithm 5 presents a simplified implementation for the ITE operation for BDDs. The algorithm takes three references to BDD nodes (entries in index) as input, named b_I, b_T, and b_E, that represent the three operands to be considered (resp. "if", "then", and "else").

ITE starts by considering terminal cases (lines 2 and 3). If b_I is not a terminal node, the operation cache is consulted on line 4. This prevents duplicate work in case the result of computing ITE(b_I, b_T, b_E) has already been calculated and stored in the cache. If this is not the case, the three BDDs referred to by b_I, b_T, and b_E are asynchronously obtained (lines 6 to 9) and their cofactors determined (lines 10 to 13). Afterwards, ITE is recursively applied via task parallelism on lines 14 to 16. Finally, the resulting BDD is inserted into the hash table (line 18) and added to the operation cache (line 19).

The ITE implementation as shown in Algorithm 5 uses a number of auxiliary functions. The function High(B, x) is used to follow an edge of a given BDD node B along the variable x. More specifically, if $B.var = x$, then $B.high$ is returned, otherwise the edge is not followed and the address of B in data is returned instead. The function Low(B, x) works like High but follows the low edge instead. TopVariable($B_0, \dots, B_k$) determines the top variable of the given sequence of BDD nodes according to the global variable ordering.

Algorithm 6: Relational Product (see also Definition 2.4)

```
1  def RelProd($b_\phi, b_\psi, X$):
2      if $b_\phi = 1 \land b_\psi = 1$: return 1
3      if $b_\phi = 0 \lor b_\psi = 0$: return 0
4      $index \leftarrow$ FindInCache(RelProd, $b_\phi, b_\psi, X$)
5      if $index =$ notfound:
6          do in parallel:
7              $B_\phi \leftarrow$ GetNode($b_\phi$)
8              $B_\psi \leftarrow$ GetNode($b_\psi$)
9          $x \leftarrow$ TopVariable($B_\phi, B_\psi$)
10         $\langle b^0_\phi, b^0_\psi \rangle \leftarrow \langle$Low($B_\phi, x$), Low($B_\psi, x$)$\rangle$
11         $\langle b^1_\phi, b^1_\psi \rangle \leftarrow \langle$High($B_\phi, x$), High($B_\psi, x$)$\rangle$
12         if $x \in X$:
13             do in parallel:
14                 $B^0_\psi \leftarrow$ GetNode($b^0_\psi$)
15                 $B^1_\psi \leftarrow$ GetNode($b^1_\psi$)
16             spawn(RelProd, $b^0_\phi$, Low($B^0_\psi, x$), $X$)
17             spawn(RelProd, $b^1_\phi$, High($B^0_\psi, x$), $X$)
18             spawn(RelProd, $b^0_\phi$, Low($B^1_\psi, x$), $X$)
19             $b_{11} \leftarrow$ call(RelProd, $b^1_\phi$, High($B^1_\psi, x$), $X$)
20             $\langle b_{10}, b_{10}, b_{00} \rangle \leftarrow \langle$sync(), sync(), sync()$\rangle$
21             spawn(ITE, $b_{00}$, 1, $b_{01}$)
22             $high \leftarrow$ call(ITE, $b_{10}$, 1, $b_{11}$)
23             $low \leftarrow$ sync()
24             $index \leftarrow$ FindOrPut(MakeNode($x, high, low$))
25         else
26             spawn(RelProd, $b^0_\phi, b^0_\psi$)
27             $high \leftarrow$ call(RelProd, $b^1_\phi, b^1_\psi$)
28             $low \leftarrow$ sync()
29             $index \leftarrow$ FindOrPut(MakeNode($x, high, low$))
30         PutInCache(RelProd, $b_\phi, b_\psi, X, index$)
31     return $index$
```

Finally, the MakeNewNode(x, b_1, b_0) function defines a fresh BDD node B such that $B.var = x$, $B.high = b_1$, and $B.low = b_0$.

The actual code implementation contains several optimisations based on ideas in [39]. Notably, additional terminal cases are defined so that less tasks are generated. This leads to better performance and scalability since each task may generate multiple network calls during its execution. Also, while waiting for the parallel block (lines 6 to 9) to terminate the implementation continuously calls communicate() to handle possible incoming steal requests.

5.3 Relational Product

Algorithm 6 shows a simplified implementation of RelProd. Like in Definition 2.4 the algorithm takes two references to BDD node as input, named b_ϕ and b_ψ, as well as a set X of variables. Note that this implementation does not take a second set X' of variables as input parameter. Instead, we assume that the current-state variables (X) and next-state variables (X') are interleaved (alternated) in the global variable order. This allows us to optimise the reachability algorithm presented in Section 2 by merging RelProd with Rename, resulting in the algorithm presented here [39].

RelProd starts by considering terminal cases (lines 2 and 3) and by consulting the operation cache. If the cache does not yet contain the computational results, the BDD nodes B_ϕ and B_ψ referred to by b_ϕ and b_ψ are asynchronously retrieved (lines 6 to 8) and their top variable x is determined. If $x \notin X$ the variable is not considered (as in this case x is a next-state variable) then RelProd continues by recursively considering the cofactors of B_ϕ and B_ψ (lines 26 to 29). Otherwise, if $x \in X$ the BDD nodes referred to by $B_\psi.high$ and $B_\psi.low$ are asynchronously obtained (lines 13 to 15) and all four cofactors determined. After that, RelProd is applied on the four cofactors to encode the Rename operation (lines 16 to 20), and the resulting two BDDs are joined by disjunction (lines 21 to 23).

The code implementation of Algorithm 6 is optimised in a way similar to ITE, for example by defining extra terminal cases.

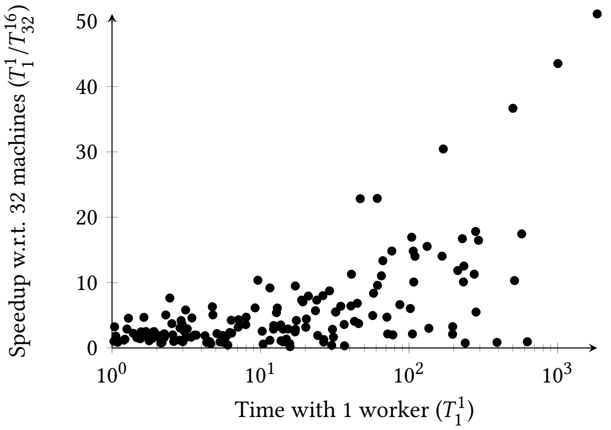
6 EXPERIMENTAL EVALUATION

We evaluated our approach by performing reachability over a large collection of models, 415 in total, taken from the well-known BEEM benchmark set [32] and from the Model Checking Contest 2016 (MCC'2016) problem set [22]. To increase performance, all these models have a partitioned transition relation [39]. The experiments have been performed on the DAS-5 compute cluster [3], thereby using up to 32 nodes, each having 16 CPU cores (Intel E5-2630v3) and 64 GB internal memory, connected via a 48 Gb/s Mellanox Infiniband network. Every node runs CentOS 7.1.1503 with kernel version 3.10.0. To measure scalability along the number of machines, each model has been benchmarked in multiple configurations (five in total), ranging from single-machine sequential runs to executions with 32 machines, 16 threads each. We repeated each experiment at least three times and considered the average measurements, which resulted in more than 18.800 experimental runs. The entire benchmarking effort took roughly one month in total.

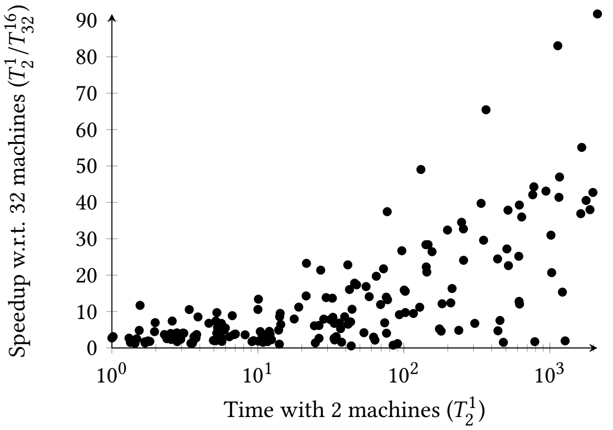
The BDD algorithms and the shared data structures have all been implemented in Berkeley UPC, version 2.22.3. The full source code and all experimental data can be found at: https://github.com/utwente-fmt/distbdd-spin17.git. We compiled the implementation using the command options `upcc -network=mxm -opt -O`, thereby employing Infiniband verbs (one-sided RDMA) as well as enabling the Mellanox Messaging Accelerator (MXM) [2] library.

We use T^t_m to denote the average measured wall-clock time in a configuration with m machines, each having t parallel threads (so $m \times t$ threads in total). We also compare our implementation with Sylvan, the current state-of-the-art parallel BDD package [40]. With S_t we denote the average measured wall-clock time using Sylvan with t workers. In every run we used a time-out of 3600 seconds and we limited memory usage to 16 GB per machine.

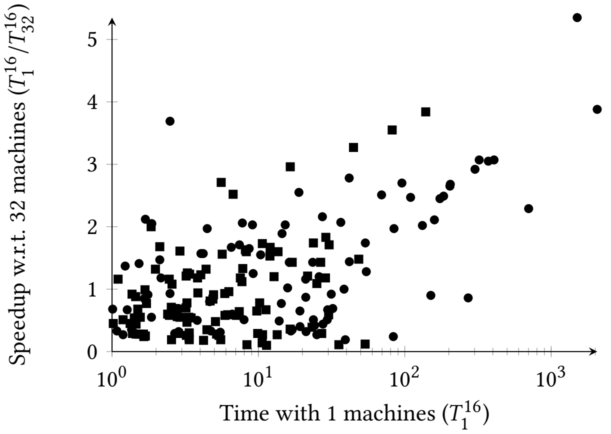
For every single-machine run we switched to quadratic probing as the collision strategy for the unique table. For parallel runs we found quadratic probing to be more efficient, since linear probing often introduces at least some clustering. However, for distributed runs the effects of clustering are outweighed by the performance gains of the dynamic range sizes.



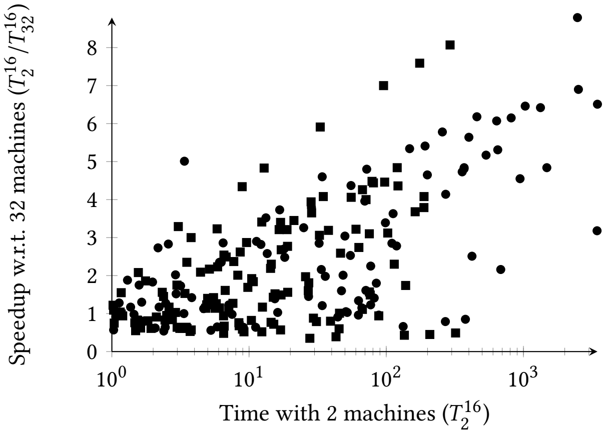
(a) Comparison between single-threaded runs (T_1^1) and speedup with respect to 32-machine distributed runs (T_{32}^{16}). Only BEEM models are considered.



(b) Comparison between two-machine distributed runs, each using one thread (T_2^1) and the speedup with respect to 32-machine distributed runs (T_{32}^{16}). Only BEEM models are considered.



(c) Comparison between T_1^{16} (16-threaded parallel runs) and speedup with respect to T_{32}^{16} (32-machine distributed runs). BEEM models are depicted as circles and Petri nets as squares.



(d) Comparison between T_2^{16} (2-machine distributed runs) and speedup with respect to T_{32}^{16} (32-machine distributed runs). BEEM models are depicted as circles and Petri nets as squares.

Figure 3: Experimental results of performing reachability over all BEEM models and Petri nets that did not time out. These four comparisons show that large model sizes correlate to better performance, both parallel (left) and distributed (right).

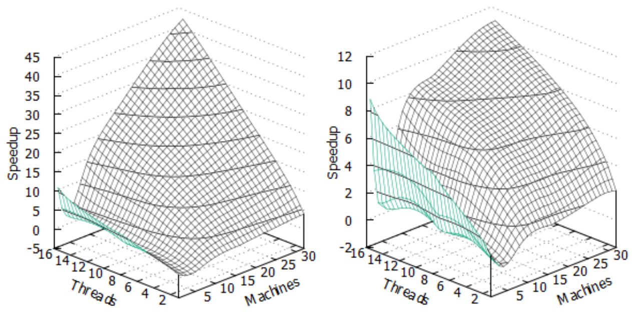


Figure 4: Speedups of performing reachability over two BEEM models: `adding.5` (left) and `anderson.8` (right). The speedups are calculated as T_1^1/T_m^t by scaling along the number m of machines and the number t of threads per machine.

6.1 Experimental Results

Figure 3 shows speedup graphs that relate the sizes of models to parallel and distributed performance. Note that Figures 3a and 3b only contain BEEM models. Since the Petri net models are generally much larger than BEEM models we only considered parallel (16-threaded) configurations while benchmarking with Petri nets.

The lowest speedups are obtained by the models with the smallest state spaces. This is because the amount of computation performed per fixed point iteration is small, which negatively impacts scalability since threads implicitly synchronise after each iteration. The largest models scale best since each iteration involves a fair amount of computational work, enough to keep all threads occupied.

Figure 3a shows that many of the larger BEEM models reach speedups of 10x to 20x compared to single-threaded runs. The largest speedup is obtained by `adding.6`, namely 51.1x. Figure 3c

Table 1: A selection of our experimental results, showing computation times and speedup of both small and very large models. A comparison with the parallel BDD package Sylvan is also shown. We use *time* to denote a timeout, *mem* to denote an "out-of-memory" failure, and "-" means that the result could not be calculated (as a result of either *time* or *mem*).

Experiments			Time			Sylvan		Speedup			
Model	# States	# Nodes	T_1^{16}	T_2^{16}	T_{32}^{16}	S_1	S_{16}	T_1^{16}/T_{32}^{16}	T_2^{16}/T_{32}^{16}	S_1/T_{32}^{16}	S_{16}/T_{32}^{16}
`2D8gradient-5x5.150`	6.8×10^{18}	3.9×10^4	*mem*	3440, 84	1083, 67	*time*	*time*	-	3.18x	-	-
`adding.6`	7.6×10^6	3.5×10^5	139, 06	291, 94	36, 19	405, 65	33, 74	3, 84x	8.07x	11, 21x	0, 93x
`anderson.8`	5.4×10^8	2.8×10^5	28, 73	75, 30	24, 32	61, 70	5, 90	1, 18x	3.10x	2, 54x	0, 24x
`armCacheCoherence`	3.2×10^8	8.4×10^3	173, 90	400, 30	71, 03	1613, 06	107, 33	2, 45x	5, 64x	22, 71x	1, 51x
`at.5`	3.2×10^7	2.3×10^5	12, 03	28, 33	7, 20	25, 42	2, 20	14, 82x	3, 93x	1, 67x	0, 31x
`blocks.4`	1.0×10^8	4.3×10^6	*time*	*time*	67, 11	963, 33	79, 51	-	-	1, 67x	1, 18x
`bridgeAndVehicles.V20`	9.8×10^6	7.5×10^5	372, 29	649, 71	122, 25	2032, 39	155, 28	3, 05x	5, 31x	16, 62x	1, 27x
`csrepetitions.4`	3.1×10^{13}	2.2×10^5	*mem*	944, 65	207, 46	2403, 42	273, 31	-	4, 55x	11, 58x	1, 32x
`drivingphils.4`	2.7×10^8	5.4×10^6	*mem*	*mem*	229, 71	2637, 96	223, 09	-	-	11, 48x	0, 97x
`erk.000100`	1.6×10^{10}	3.7×10^6	*time*	*time*	2837, 10	*time*	*time*	-	-	-	-
`FMS.100`	9.8×10^{18}	1.2×10^5	*time*	*time*	1264, 51	*time*	*time*	-	-	-	-
`gallocres.5`	1.1×10^8	6.8×10^5	1506, 70	2476, 16	281, 80	*time*	415, 21	5, 35x	8, 79x	-	1, 47x
`kanban.50`	1.0×10^{16}	1.1×10^5	2063, 76	3462, 97	532, 09	*time*	2714, 06	3, 88x	6, 51x	-	5, 10x
`lifts.8`	1.2×10^7	1.6×10^6	*mem*	*time*	43, 74	301, 96	24, 41	-	-	6, 90x	0, 56x
`philosophers.100`	1.2×10^{19}	2.7×10^4	83, 53	269, 80	341, 60	*time*	1150, 70	0, 24x	0, 79x	-	3, 37x
`philosophers.200`	1.2×10^{19}	1.2×10^5	*mem*	*mem*	3089, 55	*time*	*time*	-	-	-	-
`sg-2-1-2`	2.1×10^{12}	4.4×10^5	*mem*	*mem*	276, 96	*time*	*time*	-	-	-	-
`soli1`	1.9×10^8	2.0×10^7	*mem*	1030, 26	159, 41	1871, 21	*time*	-	6, 46x	11, 74x	-
`swimmingpool.5`	5.9×10^8	2.2×10^6	*mem*	2516, 12	364, 89	*time*	512, 91	-	6, 90x	-	1, 41x
`swimmingpool.6`	1.7×10^9	3.7×10^6	*time*	*time*	868, 60	*time*	1742, 65	-	-	-	2, 01x
`swimmingpool.7`	4.2×10^9	6.6×10^6	*mem*	*time*	2563, 81	*time*	*time*	-	-	-	-
`telephony.6`	1.5×10^9	9.7×10^5	*mem*	*mem*	91, 24	1222, 55	104, 58	-	-	13, 40x	1, 15x
`tokenring.20`	2.4×10^{10}	1.2×10^6	*mem*	*time*	614, 06	*time*	2810, 64	-	-	-	4, 58x

shows the speedup of a 32-machine configuration with respect to parallel runs, thereby including the larger Petri net models. To the best of our knowledge, we present the first distributed BDD package that achieves speedup compared to parallel executions.

Compared to single-machine runs the performance drops significantly when two machines are used. This is because with two machines the number of threads employed is insufficient to compensate for the loss of data-locality (that is, for the use of the network). By increasing the number of threads the communicational overhead is compensated and the performance increases compared to sequential runs. This is illustrated in Figure 4, in which we highlight two BEEM models, namely `adding.5` and `anderson.8` (the other models show a similar pattern). In both models the performance drops when two machines are used. However, the performance increases again when more machines and threads are added. Figures 3b and 3d show the scalability of distributed runs relative to a setting with two machines (in which the Infiniband network is actively used). Likewise to Figure 4, both graphs show that the performance increases when machines are added.

Table 1 gives statistical information of a selection of our experimental results and compares it with the parallel performance of Sylvan. Observe that none of the Sylvan runs terminated due to a lack of memory. Instead, Sylvan applies garbage collection whenever needed and this eventually results in a time-out when memory runs short. Our implementation does not yet support garbage collection; the algorithm terminates when the unique table is full. We plan to implement garbage collection in future work.

Table 1 shows that the efficiency of Sylvan is better than the efficiency of our implementation. This is however expected; Sylvan is heavily optimised to maximise parallel performance in NUMA architectures. Our focus was not to optimise for single-machine parallelism, but rather to propose a scalable distributed implementation. In future work we aim to combine both approaches.

Nonetheless, we see that the larger models benefit from the extra memory available in a 32-machine setting. For example, we observed that some models (e.g. `kanban.50` and `tokenring.20`) perform better than Sylvan with 16 threads. Other benchmark models, like `erk.000100` and `FMS.100`, could be processed by our implementation but not with Sylvan due to the lack of available memory. This shows that our implementation can be more efficient than Sylvan when processing very large models, which is a clear improvement on previous work on distributed symbolic reachability.

7 CONCLUSION

We presented new algorithms for BDD-based symbolic reachability, targeting distributed/multi-core systems, and investigated both their parallel and distributed speedup. We carefully redesigned the basic components for scalable BDD operations: a distributed hash table and private-deque hierarchical work stealing. The main challenge in their design was to minimise the overhead of network communication. Compared to existing work on distributed hash tables and load balancers, we proposed *lockless* implementations designed to minimise the number of networking operations. Moreover, all components have been designed to take full advantage of the newest networking technology. The BDD operations themselves are designed to effectively use the idle-times of processes and attempt to overlap computation with communication as much as possible. To our knowledge, we present the first BDD package that targets both distributed and multi-core architectures.

We investigated the parallel and distributed speedups on a compute cluster using 32 machines, each using up to 16 threads, connected via an Infiniband network. We obtained speedups up to 51.1x, thereby improving over existing work. As far as we know these are higher than any speedups reported for symbolic reachability in previous literature. Our implementation outperforms Sylvan when memory runs short and can very well be used when the state space does not fit into the memory of a single machine.

We are currently planning to implement garbage collection and to integrate our algorithms into the LTSMIN toolset [21]. Moreover, there are still many open questions concerning efficiency improvements. For example, the idea of speculative computations [9] may also apply to our approach. Also, the scalability of alternative decision diagrams on distributed hardware is yet unknown. Finally, our ideas may extend to accelerators and to GPU clusters. Although there is support for running Berkeley UPC natively on Xeon Phi's [25] we did not yet assert its performance. Other related work on this matter includes GPU state space exploration [4, 42] which is focused on explicit-state graph searching.

REFERENCES

[1] 2008. Infiniband Trade Association. (2008). http://www.infinibandta.org
[2] 2015. Mellanox's Messaging Accelerator. (2015). http://www.mellanox.com/vma
[3] 2015. The Distributed ASCI Supercomputer 5. (2015). http://www.cs.vu.nl/das5
[4] J. Barnat, P. Bauch, L. Brim, and M. Češka. 2012. Designing fast LTL model checking algorithms for many-core GPUs. In *Journal of Parallel and Distributed Computing*, Vol. 72. Elsevier, 1083–1097.
[5] R.D. Blumofe et al. 1995. *Cilk: An Efficient Multithreaded Runtime System*. Vol. 30. IEEE Computer Society. 356fi?!368 pages.
[6] R.E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, Vol. 35. 677–691.
[7] J. Chen and P. Banerjee. 1999. Parallel Construction Algorithms for BDDs. In *ISCAS*. IEEE, 318–322.
[8] M. Chung and G. Ciardo. 2004. Saturation NOW. In *Quantitative Evaluation of Systems*. IEEE, 272–281.
[9] Ming-Ying Chung and Gianfranco Ciardo. 2011. Speculative Image Computation for Distributed Symbolic Reachability Analysis. *Journal of Logic and Computation* (2011), 63–83.
[10] G. Ciardo, Y. Zhao, and X. Jin. 2009. Parallel symbolic state-space exploration is difficult, but what is the alternative?. In *PDMC*.
[11] C. Coarfa et al. 2005. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *PPoPP*. ACM, 36–47.
[12] H. Cohen, J. Whaley, J. Wildt, and N. Gorogiannis. BuDDy. (????). http://sourceforge.net/p/buddy
[13] J. Dinan et al. 2008. Scioto: a Framework for Global-View Task Parallelism. In *Parallel Processing*. IEEE, 586–593.
[14] J. Dinan et al. 2009. Scalable Work Stealing. In *High Performance Computing Networking, Storage and Analysis*. ACM, 53.
[15] A. Dragojević, A. Narayanan, O. Hodson, and M. Castro. 2014. Farm: Fast Remote Memory. In *11th USENIX Conference on Networked Systems Design and Implementation, NSDI*, Vol. 14.
[16] J. Ezekiel, G. Lüttgen, and G. Ciardo. 2007. Parallelising Symbolic State-Space Generators. In *CAV*. Springer, 268–280.
[17] K. Faxén. 2009. Wool - a Work Stealing Library. In *SIGARCH*. 93–100.
[18] K. Faxén. 2010. Efficient Work Stealing for Fine Grained Parallelism. In *Parallel Processing (ICPP)*. IEEE, 313–322.
[19] O. Grumberg, T. Heyman, T. Ifergan, and A. Schuster. 2005. Achieving Speedups in Distributed Symbolic Reachability Analysis through Asynchronous Computation. In *Correct Hardware Design and Verification Methods*. Springer, 129–145.
[20] O. Grumberg, T. Heyman, and A. Schuster. 2006. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design* (2006), 157–175.
[21] G. Kant et al. 2015. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS*. Springer, 692–707.
[22] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. 2016. Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php. (June 2016).
[23] A. Laarman, J. van de Pol, and M Weber. 2010. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *Conference on Formal Methods in Computer-Aided Design*. FMCAD, 247–256.
[24] A. Lovato, D. Macedonio, and F. Spoto. 2014. A Thread-Safe Library for Binary Decision Diagrams. In *SEFM*. Springer, 35–49.
[25] M. Luo et al. 2013. UPC on MIC: Early Experiences with Native and Symmetric Modes. In *Conference on PGAS Programming Models*. 198–2010.
[26] K. Milvang-Jensen and A.J. Hu. 1998. BDDNOW: a Parallel BDD Package. In *Formal Methods in Computer-Aided Design*. Springer, 501–507.
[27] S. Min, C. Iancu, and K. Yelick. 2011. Hierarchical Work Stealing on Manycore Clusters. In *5th Conference on Partitioned Global Address Space Programming Models*.
[28] S. Minato, N. Ishiura, and S. Yajima. 1990. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Design Automation Conference*. IEEE, 52–57.
[29] C. Mitchell, Y. Geng, and L. Jinyang. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*. 103–114.
[30] W. Oortwijn, T. van Dijk, and J. van de Pol. accepted 2015. A Distributed Hash Table for Shared Memory. In *Parallel Processing and Mathematics*.
[31] Y. Parasuram, E. Stabler, and S. Chin. 1994. Parallel Implementation of BDD Algorithms Using a Distributed Shared Memory. In *System Sciences*, Vol. 1. IEEE, 16–25.
[32] R. Pelánek. 2007. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software*. Springer, 263–267.
[33] D. Sahoo et al. 2005. Multi-Threaded Reachability. In *42nd annual Design Automation Conference*. ACM, 467–470.
[34] F. Somenzi. 1998. CUDD: CU decision diagram package release 2.5.0. *University of Colorado at Boulder* (1998).
[35] T. Stornetta and F. Brewer. 1996. Implementation of an Efficient Parallel BDD Package. In *Design Automation Conference*. ACM, 641–644.
[36] T. Szepesi, B. Wong, B. Cassell, and T. Brecht. 2014. Designing a Low-Latency Cuckoo Hash Table for Write-Intensive Workloads Using RDMA. In *First International Workshop on Rack-scale*.
[37] Tom van Dijk. 2016. *Sylvan: multi-core decision diagrams*. Ph.D. Dissertation. Enschede. http://doc.utwente.nl/100676/ IPA dissertation series no. 2016-09.
[38] T. van Dijk and J. van de Pol. 2014. Lace: Non-Blocking Split Deque for Work-Stealing. In *7th International Euro-Par Workshop on Multi-/Many-core Computing Systems*. Springer International Publishing, 206–217.
[39] T. van Dijk and J. van de Pol. 2015. Sylvan: Multi-core Decision Diagrams. In *TACAS*. Springer, 677–691.
[40] T. van Dijk et al. 2015. A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking. In *Dependable Software Engineering: Theories, Tools, and Applications*. 35–51.
[41] D.B. Wagner and B.G. Calder. 1993. Leapfrogging: a Portable Technique for Implementing Efficient Futures. In *PPOPP*. 208–217.
[42] A. Wijs and D. Bošnački. 2015. Many-core on-the-fly model checking of safety properties using GPUs. In *International Journal on Software Tools for Technology Transfer*. Springer, 1–17.