Parallel Disk-Based Computation for Large, Monolithic Binary Decision Diagrams

Daniel Kunkle* Vlad Slavici Gene Cooperman*

Northeastern University
360 Huntington Ave.
Boston, Massachusetts 02115
{kunkle,vslav,gene}@ccs.neu.edu

ABSTRACT

Binary Decision Diagrams (BDDs) are widely used in formal verification. They are also widely known for consuming large amounts of memory. For larger problems, a BDD computation will often start thrashing due to lack of memory within minutes. This work uses the parallel disks of a cluster or a SAN (storage area network) as an extension of RAM, in order to efficiently compute with BDDs that are orders of magnitude larger than what is available on a typical computer. The use of parallel disks overcomes the bandwidth problem of single disk methods, since the bandwidth of 50 disks is similar to the bandwidth of a *single* RAM subsystem. In order to overcome the latency issues of disk, the Roomy library is used for the sake of its latency-tolerant data structures. A breadth-first algorithm is implemented. A further advantage of the algorithm is that RAM usage can be very modest, since its largest use is as buffers for open files. The success of the method is demonstrated by solving the 16-queens problem, and by solving a more unusual problem — counting the number of tie games in a three-dimensional 4×4×4 tic-tac-toe board.

Categories and Subject Descriptors: I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms — *Algebraic algorithms*, *Analysis of algorithms*; E.1 [**Data Structures**]: *Distributed data structures*

General Terms: Algorithms, Experimentation, Performance

Keywords: parallel, disk-based, binary decision diagram, BDD, breadth-first algorithm

1. INTRODUCTION

There are three widespread symbolic techniques for formal verification currently in use: binary decision diagrams, SAT solvers, and explicit state model checking. Binary decision diagrams (BDDs) have a particular attraction in being able to compactly represent a Boolean function on k Boolean variables. Because BDDs are a representation of the general solution, they are an example of *symbolic model checking* — in contrast with SAT-solving and explicit state model checking, which find only particular solutions.

BDDs have found widespread use in industry — especially for circuit verification. Unfortunately, as with many formal verification methods, it tends to quickly run out of space. It is common for a BDD package to consume the full memory of RAM in a matter of minutes to hours.

To counter this problem of storage, a novel approach is demonstrated that uses parallel disks. While we are aware of approaches using the local disks of one computer and of approaches using the RAM of multiple computers or CPUs, we are not aware of work simultaneously using the parallel disks of many computers. The parallel disks may be the local disks of a cluster, or the disks of a SAN in a cluster.

The parallelism is not for the sake of speeding up the computation. It is only to support the added storage of parallel disks. Because aggregate disk bandwidth is the bottleneck of this algorithm, more disks produce a faster computation. Nevertheless, the goal of the algorithm is to extend the available storage — *not* to provide a parallel speedup beyond the traditional sequential computation in RAM.

The new package is demonstrated by building large QBDDs (quasi-reduced BDDs) for two combinatorial problems. The first is the well-known N-queens problem, and is used for comparability with other research. The second combinatorial problem may be new in the literature. The number of tie boards are counted on a 3-dimensional tic-tac-toe board of dimensions $4 \times 4 \times 4$.

The question of comparability of this new work with previous work is a subtle one. Previously, researchers have used the disks of a single node in an effort to extend the available storage of a sequential BDD package [2, 21, 24, 25]. Researchers have also developed parallel BDD algorithms with the goal of making BDD computations run faster [11, 15, 19, 22, 28, 29]. All of the work cited was performed in the 1990s, with little progress since then. One exception is that in 2001, Minato and Ishihara [16] demonstrated a streaming BDD mechanism that incorporates pipelined parallelism. That work stores its large result on a single disk at the end of the pipeline, but it does not use parallel disks.

Given that there has been little progress in disk-based and in parallel BDDs in the last ten years, there is a question of how to produce a fair scalability comparison that takes into account the advances in hardware over that time. While the last ten years have seen only a limited growth in CPU speed,

*This work was partially supported by the National Science Foundation under Grant CNS-0916133.

PASCO 2010, 21–23 July 2010, Grenoble, France.

they have seen a much greater growth in the size of RAM.

We use the N-queens problem to validate the scalability of our new algorithm. We do this in two ways. Recall that the N-queens problem asks for the number of solutions in placing N non-attacking queens on an $N \times N$ chess board.

In the first validation, we compare with a traditional sequential implementation on a computer with large RAM. The BuDDy package [8] was able to solve the N-queens problem for 14 queens or less, when given 8 GB of RAM. When given 56 GB of RAM, BuDDy was still not able to go beyond 14 queens. In comparison, the new program was able to solve the 16-queens problem. One measure of the size of the computation is that the size of the canonical BDD produced for 16 queens was 30.5 times larger than that for 14 queens. The number of solutions with 16 queens was 40.5 times the number of solutions with 14 queens.

A second validation of the scalability is a comparison with the more recent work of Minato and Ishihara [16]. That work used disk in 2001 to store a BDD solution to the N-queens problem. Like BuDDy, that work was also only able to solve the 14-queens problem. Nevertheless, it represented the state of the art for using BDDs to solve the N-queens problem at that time. It reported solving the 14-queens problem in 73,000 seconds with one PC and in 2,500 seconds with 100 PCs. The work of this paper solves the 16-queens problem using 64 computers in 100,000 seconds. The ROBDD for 16 queens is 30 times larger than that for 14 queens. The bandwidth of disk access has not grown by that factor of 30 over those 9 years.

To our knowledge, the BDD package presented here is the first demonstration of the practicality of a breadth-first *parallel disk-based computation*. Broadly, parallel-disk based computation uses disks as the main working memory of a computation, instead of RAM. The goal is to solve space-limited problems without significantly increasing hardware costs or radically altering existing algorithms and data structures. BDD computations make an excellent test of parallel-disk based computation because larger examples can exhaust several gigabytes of memory in just minutes (see experimental results in Section 5).

There are two fundamental challenges in using disk-based storage as main memory:

> **Bandwidth**: roughly, the bandwidth of a single disk is 50 times less than that of a single RAM subsystem (100 MB/s versus 5 GB/s). Our solution is to use many disks in parallel, achieving an aggregate bandwidth comparable to RAM.
>
> **Latency**: worse than bandwidth, the latency of disk is many orders of magnitude worse than RAM. Our solution is to avoid latency penalties by using streaming data access, instead of costly random access.

Further, as an optimization, we employ a hybrid approach: using serial, RAM-based methods for relatively small BDDs; transitioning to parallel disk-based methods for large BDDs.

Briefly, the main points of novelty of our approach are:

- This is the first package to efficiently use both parallel CPUs and parallel disks. More disks make the computation run faster. The package also allows computations for monolithic BDDs larger than ever before (the alternative approach is partitioning the very large BDD into smaller BDDs). There are advantages to using one monolithic BDD rather than many sub-BDDs: the duplicate work is reduced, thus reducing computation time in some cases; sometimes, partitioning a BDD requires domain-specific knowledge, while our approach does not. There are, however, good partitioning methods for which no domain-specific knowledge is necessary. In any case, our approach can be combined with BDD partitioning, to solve very very large problems.

- By using Roomy [12], the high-level algorithms are separated from the low-level memory management, garbage collection and load balancing. Most existing BDD packages have at least some of these three components integrated in the package, while our BDD package deals only with the "application" level, thus making it very flexible.

- As opposed to most other packages for manipulating very large BDDs, we do not require that a level in a BDD fit entirely in the aggregate RAM of a cluster. The RAM is used primarily to hold buffers for open files. While the number of open files can potentially grow in some circumstances, a technique analogous to the use of two levels in external sorting can be employed to again reduce the need for RAM, at a modest slowdown in performance (less than a factor of two).

Related work is described in Section 1.1. An overview of binary decision diagrams is presented in Section 2. A brief overview of the Roomy library is given in Section 3. Section 4 presents the parallel algorithms that are the foundation of this work. Section 5 presents experimental results. This is followed by a discussion of future work in Section 6.

1.1 Related Work

Prior work relevant to the proposed BDD package comes from two lines of research: sequential breadth-first BDD algorithms and parallel RAM-based BDD algorithms. The two approaches often overlap, resulting is parallel RAM-based breadth-first algorithms. However, there are no prior successful attempts at creating a parallel disk-based package in the literature.

Algorithms and representations for large BDDs stored in secondary memory have been the subject of research since the early 1990s. Ochi et al. [21] describe efficient algorithms for BDDs too large to fit in main memory on the commodity architectures available at that time. The algorithms are based on breadth-first search, thus avoiding random pointer chasing on disk. Shared Quasi-Reduced BDDs (SQBDDs) are used to minimize lookups at random locations on disk. These algorithms are inherently sequential, requiring the use of a local "operation-result-table". Random lookups to this local table avoid the creation of duplicate nodes.

Ashar and Cheong [2] build upon the ideas of [21] and improve the performance of BDD operations based on breadth-first search by removing the need for SQBDDs and using the most compact representation, the Reduced Ordered BDD (ROBDD). Duplicate avoidance is performed by lookups to local queues, which make the algorithm hard to parallelize.

A High Performance BDD Package which exploits the memory hierarchy is presented in [24]. The package builds upon the findings in [2], making the algorithms more general and more efficient. Even though [24] introduces concepts as

"superscalarity" and "pipelining" for BDD algorithms, the presented package is still an inherently sequential one, relying on associative lookups to request queues.

Ranjan et al. [22] propose a BDD package based on breadth-first algorithms for a network of workstations, making possible the use of distributed memory for large BDD computations. However, their algorithms are sequential – the computation is carried out one workstation at a time, thus only taking advantage of the large amount of space that a network of workstations offers, without parallelizing the processing of data. Their approach is infeasible for efficient parallelization.

Stornetta and Brewer [28] propose a parallel BDD package for distributed fast-memory (RAM) based on depth-first algorithms, while providing a mechanism for efficient load balancing. However, their algorithms incur a large communication overhead. Any approach based on depth-first search would be infeasible for a parallel disk-based package, because the access pattern has a high degree of randomness. Milvang-Jensen and Hu [15] provide a BDD package building upon ideas in [22]. As opposed to [22], the package proposed in [15] is parallel – the workstations in a network all actively process data at the same time. However, since each workstation deals only with a sequence of consecutive levels, the workload can become very unbalanced, which would be unacceptable for a parallel disk-based BDD algorithm.

In 1997, Yang and O'Hallaron [29] propose a method for parallel construction of BDDs for shared memory multiprocessors and distributed shared memory (DSM) systems based on their technique named *partial breadth-first expansion*, having as starting point the depth-first/breadth-first hybrid approach in [1]. Their algorithms rely on very frequent synchronization of global data structures and on exclusive access to critical sections of the code which make this approach infeasible for any distributed-memory system. Also in 1997, Bianchi et al. [3] propose a MIMD approach to parallel BDDs. Their approach exhibits very good load balancing, but the communication becomes a bottleneck, making it infeasible to be adapted for parallel disks, because it does not delay and batch requests.

Other methods for manipulating very large BDDs are partitioning the BDD by means of partitioning the Boolean space using window functions [18] or by determining good splitting variables [6].

The newest result mentioned so far in this section is from 2001. Although there is substantial BDD-related work after 2001, little of it is concerned with providing fundamentally different ways of parallelizing BDD algorithms. Most use existing ideas and improve upon them or optimize the implementations. Most BDD-related research in the past decade was oriented towards better and faster algorithms for reachability analysis [9, 10, 23], compacting the representation of BDDs [27] or variable ordering [7, 17, 20].

There are many other methods for improving the efficiency of BDD processing. One such method, described by Mao and Wu [14], applies Wu's method to symbolic model checking.

2. BACKGROUND: BINARY DECISION DIAGRAMS

The conceptual idea behind Binary Decision Diagrams is quite simple. One wishes to represent an arbitrary Boolean function from k Boolean variables to a Boolean result. For example, "and", "or", and "xor" are three functions, each of which is a Boolean function on two Boolean variables. In the following description, the reader may wish to refer to Figure 1 for an example.

An *ordered binary decision diagram* (OBDD) fixes an ordering of the k Boolean variables, and then represents a binary decision tree with respect to that ordering. The tree has $k+1$ levels, with each of the first k levels corresponding to the next Boolean variable in the ordering. For each node at level i, the Boolean values of $x_1, \ldots, x_{i-1}$ are known, and the child along the left branch of a tree represents those same Boolean values, along with x_i being set to 0 or false. Similarly, the descendant of a node at level i along the right branch of a tree corresponds to setting x_i to 1 or true. For a node at level $k+1$, the Boolean values of $x_1, \ldots, x_k$ at that node are all determined by the unique path from the root of the tree to the given node. Next, a node at level $k+1$ is set either to false or to true, according to the result of the Boolean function being represented.

For example, the "and" function on two variables will have three levels: one root node at the first level, two nodes at the second level, and four leaf nodes at the third level. Three of the leaf nodes are set to false and one is set to true.

A *reduced ordered binary decision diagram* (ROBDD) is a lattice representing the same information as an OBDD, but with maximal sharing of nodes. More precisely, one can progressively convert an OBDD into an ROBDD as follows. First, identify all "true" nodes of the OBDD with a single "true" node of the ROBDD, and similarly for the "false" nodes. Thus, there are exactly two nodes at level $k+1$ in the ROBDD. One says that the nodes at level $k+1$ have been *reduced*.

Next, one iterates moving bottom up. Assume that all nodes at levels $i+1, i+2, \ldots, k$ of the OBDD have been reduced. Two nodes at level i are *reduced*, or identified with each other, if they have the same left child node and the same right child node. After all possible reductions at level i are completed, one continues to level $i-1$, and so on.

The structure described so far is in fact a *quasi-reduced BDD* (QBDD), which has no redundant nodes, and a child of a node can be either a node at the next level or a terminal. A fully reduced BDD (ROBDD) has one additional optimization: nodes that have identical left and right children (sometimes called *forwarding nodes*) are removed. So, children can be from any lower level.

Often, one refers to an ROBDD as simply a BDD for short. The advantage of a BDD is a compact representation. For example, the parity Boolean function is the Boolean function that returns true if an even number of the k input variables are true, and it returns false otherwise. A BDD representing the parity function will have exactly two distinct nodes for each level below level 1. Intuitively, one of the two nodes at level i can be thought of as indicating that the variables $x_1, \ldots, x_{i-1}$ have even parity, and the other node represents odd parity. Following the branch that sets x_i to true will lead from a node at level i to a node of the opposite parity at level $i+1$. Following the branch that sets x_i to false will lead to a node of the same parity at level $i+1$.

Finally, to take the logical "and" of two BDDs, one constructs a new BDD in top-down fashion using the Shannon expansion. If one wishes to form the "and" of two nodes at the same level (one from the first BDD and one from the second BDD), the result will be a new node whose left child

will be the "and" of the two original left children, and the right child will be the "and" of the two original right children. Thus, to construct the "and" of two nodes requires that one recursively construct the "and" of the left children along with the "and" of the right children. The recursion stops when reaching the leaf nodes. The combinatorial explosion is reduced (but not necessarily eliminated) by using an implementation related to dynamic programming [4].

Logical "or" and "xor" of BDDs are implemented similarly to logical "and". The generic algorithm for combining two BDDs is often called `apply` in the literature. Logical "not" of a BDD consists of exchanging the values of the two leaf nodes, "true" and "false".

The QBDD is preferred over the ROBDD due to its more natural specification of node levels. In a QBDD, the *level* of a node is the number of edges that have to be traversed on any path from the root to the node. Note that the level definition is consistent for any non-terminal node – any path from the root to a node in a QBDD has the same length. We use the convention that the root of a BDD is at level 0. Level i corresponds to variable index i.

Figure 1 shows a comparison between OBDD, QBDD and ROBDD representations of the same Boolean formula.

3. A BRIEF OVERVIEW OF ROOMY

This section gives a brief overview of the parts of Roomy that are pertinent to the BDD algorithms in Section 4. Complete documentation and source code for Roomy can be found online [12]. Also published in the same proceedings as this paper is a tutorial on Roomy [13], which describes the Roomy programming model and gives example programs.

We chose to use Roomy for our implementation because it removes the need for the programmer to deal with various correctness and performance issues that arise when writing parallel disk-based applications.

3.1 Goals and Design of Roomy

Roomy is implemented as an open-source library for C/C++ that provides programmers with a small number of simple data structures (arrays, unordered lists, and hash tables) and associated operations. Roomy data structures are transparently distributed across many disks, and the operations on these data structures are transparently parallelized across the many compute nodes of a cluster. All aspects of the parallelism and remote I/O are hidden within the Roomy library.

The primary goals of Roomy are:

1. to provide the most general programming model possible that still biases the application programmer toward high performance for the underlying parallel disk-based computation.

2. the use of full parallelism; providing not only the use of parallel disks (e.g., as in RAID), but also parallel processing.

3. to allow for a wide range of architectures, for example: a single shared-memory machine with one or more disks; a cluster of machines with locally attached disks; or a compute cluster with storage area network (SAN).

The overall design of Roomy has four layers: foundation; programming interface; algorithm library; and applications.

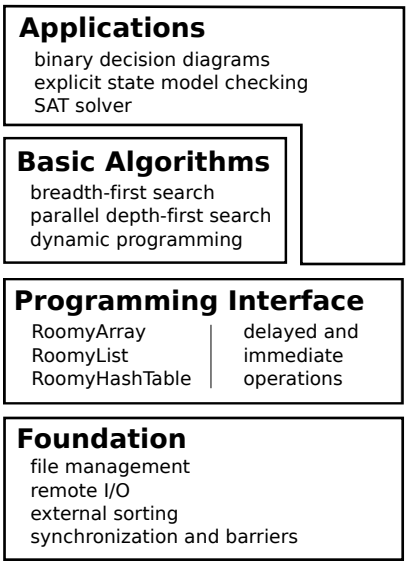


Figure 2: The layered design of Roomy.

Figure 2 shows the relationship between each of these layers, along with examples of the components contained within each layer.

3.2 Roomy-hashtable

Because the Roomy-hashtable is the primary data structure used in the parallel disk-based BDD algorithms in Section 4, it is used to illustrate the various operations provided by the Roomy data structures. Operations for the Roomy-array and Roomy-list are similar in nature.

A *Roomy-hashtable* is a variable sized container which maintains a mapping between user-defined *keys* and *values*. Those elements can be built in data types (such as integers), or user defined structures of arbitrary size. A Roomy-hashtable can be of any size, up to the limits of available disk space, and will grow as needed as elements are inserted.

Each Roomy-hashtable is stored as a number of RAM-sized *subtables*, distributed across the disks of a cluster (or the disks of a SAN). Delayed operations are buffered to disk and co-located with the subtable they reference. Once many delayed operations are collected, each subtable can be processed independently to complete the delayed operations, avoiding costly random access patterns.

Below are the operations provided by a Roomy-hashtable, categorized by whether they are delayed or immediate.

Roomy-hashtable delayed operations..

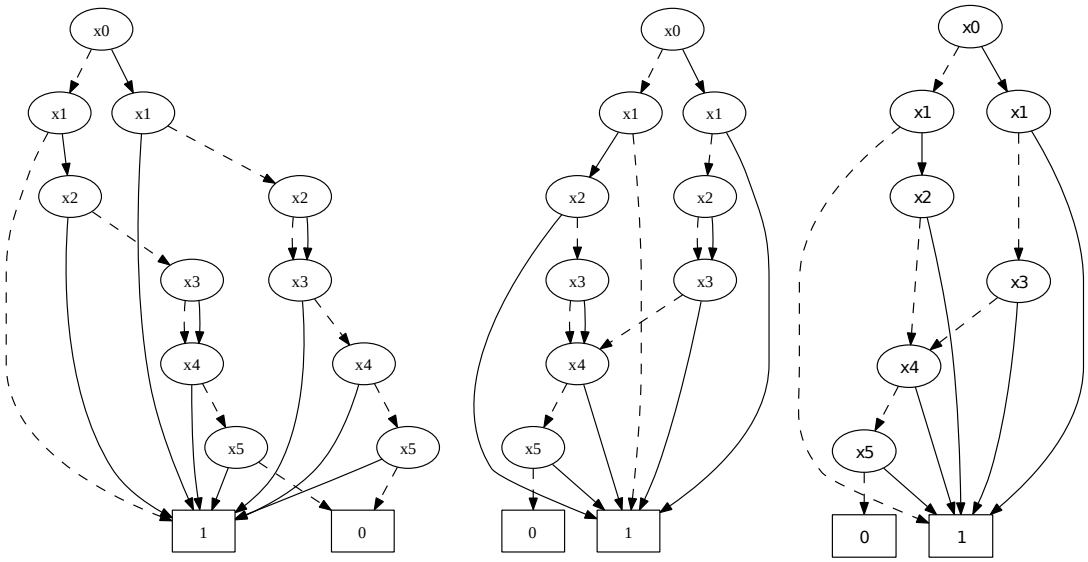
`insert`: insert a given key/value pair, replacing any existing pair with the same key

`access`: apply a user defined function to the key/value pair with the given key, if it exists

`remove`: remove the key/value pair with the given key, if it exists

Roomy-hashtable immediate operations..

`size`: return the number of key/value pairs stored in the hashtable



(a) OBDD representation (b) QBDD representation (c) ROBDD representation

Figure 1: OBDD, QBDD and ROBDD representations for Boolean formula $(x_0 \vee \neg x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (\neg x_0 \vee x_3 \vee x_1 \vee x_4 \vee x_5)$. Solid lines represent the high (true) branch, dashed lines the low (false) branch. The OBDD is the least compact representation. Combining identical nodes (for variables x_4 and x_5 in this case) will create a QBDD. The QBDD can be further reduced to an ROBDD by removing a node if it has identical high and low pointers (for variables x_2 and x_3 in this case).

`map`: applies a user defined function to each key/value pair in the hashtable

`reduce`: applies a user defined function to each key/value pair in the hashtable and returns a value (e.g. the sum of all of the values)

`predicateCount`: return the number of key/value pairs in the hashtable that satisfy the given predicate

`sync`: perform all outstanding delayed `insert`, `access`, and `remove` operations

4. PARALLEL DISK-BASED ALGORITHMS FOR BDDS

The proposed Roomy-based parallel disk-based BDD package uses the SQBDD (shared QBDD) representation described in [21]. This package does not support BDDs that are shared among multiple Boolean expressions (although this feature can easily be implemented in the package), thus making the representation a quasi-reduced BDD (QBDD). Although [2] describes cases in which the ROBDD representation can be a few times more compact than the corresponding (S)QBDD representation, this is not the case for the problems considered here. The focus of the experimental results (see Section 5) is on the solutions to combinatorial problems (the N-queens problem and 3-D tic-tac-toe) in which the inherent regularities and symmetries of the problem lead to QBDDs that are only at most 15 % larger than the corresponding ROBDD (the percentage was observed experimentally). Since most operations in Roomy are delayed and processed in batches, the locality advantage of a the QBDD representation is important (a node in a QBDD can only have as children the nodes at the immediate next level). This is especially true in cases where the QBDD representation is not much larger than the ROBDD representation, as is the case here.

Since the QBDDs for the considered problems are close in size to their corresponding ROBDDs, it means that one can solve problems one or two orders of magnitude larger using disk-based methods than by using a RAM-based alternative.

The rest of this section presents a brief implementation description of our package, followed by a high-level description of the main operations: `apply`, `any-SAT` and `SAT-count`.

4.1 Implementation Description

Each BDD is represented in a quasi-reduced form. This means that, when a node at level i has a child at level $j > i + 1$, there are $j - i - 1$ padding nodes inserted in the BDD, one at each level between $i + 1$ and $j - 1$, thus creating an explicit path from the node at level i to its child at level j. These padding nodes would be considered redundant nodes in an ROBDD implementation, and thus, would not exist. However, a QBDD needs such redundant nodes to maintain the invariant that each node at a certain level only has children at the immediate next level.

Our implementation of a QBDD maintains a data structure that contains:

- d, the depth of the BDD.
- *Nodes*, an array of d Roomy-hashtables. $Nodes[i]$ is a Roomy-hashtable that contains all the nodes at level i of the BDD in key-value form. The key is the BDD-unique node id and the value is a pair of ids *(low(id), high(id))*, which reference the children of the node. Each of the children is either at level $i + 1$ or is a terminal node.

This representation allows the *delayed* lookup of a node in the BDD by providing only its unique id. Since each

BDD level has its own Roomy-hashtable, when processing a certain level i we only need to inspect $Nodes[i]$ (*the current nodes*), $Nodes[i+1]$ (*the child nodes*) and, in some cases $Nodes[i-1]$ (*the parents*). Inspecting the entire BDD is not necessary. This is important for very large BDDs, in which even the nodes at a single level can overflow the aggregate RAM of a cluster.

4.2 The APPLY Algorithm

The foundation that most BDD algorithms are built upon is the `apply` algorithm. `apply` applies a Boolean operation (like `or`, `and`, `xor`, `implication`, a.s.o) to two BDDs. If BDDs A and B represent Boolean expressions f_A and f_B respectively, then $AB = op(A, B)$ is the BDD which represents the Boolean expression $op(f_A, f_B)$. The traditional RAM-based `apply` algorithm performs this operation by employing a depth-first algorithm, as explained in [5]. A memoization table in which already computed results are stored increases the performance of the algorithm from exponential to quadratic in the number of input nodes. The pseudo-code is presented in Algorithm 1.

Algorithm 1 RAM-based depth-first `apply`

Input: BDD A with root-node n_1, BDD B with root-node n_2, Boolean operator op
Output: n, the root node of a BDD representing $op(A, B)$

```
Init M (memoization table for new nodes)
if M(n1, n2) was already set then
  return M(n1, n2)
if n1 ∈ {0,1} AND n2 ∈ {0,1} then
  n = op(n1, n2)
else if var(n1) = var(n2) then
  n ← new Node(var(n1), apply(low(n1), low(n2), op),
               apply(high(n1), high(n2), op))
else if var(n1) < var(n2) then
  n ← new Node(var(n1), apply(low(n1), n2, op),
               apply(high(n1), n2, op))
else
  n ← new Node(var(n2), apply(n1, low(n2), op),
               apply(n1, high(n2), op))
  set M(n1, n2) ← n
return n
```

To create a version of `apply` applicable for data stored in secondary memory, [21] converts the requirements on the implicit process stack in Algorithm 1 into an explicit requirement queue. This is the main idea behind converting a DFS algorithm into a BFS one. Our parallel disk-based package uses the same framework to create an efficient parallel disk-based `apply`. The structure used is a *parallel queue*, implemented on top of a Roomy-hashtable. A parallel queue relaxes the condition that all requirements must be processed one by one to a condition that all requirements at a certain level in the BFS must be processed before any requirement at the next level. In this way, parallel processing of data at each BFS level is enabled.

An important feature that Roomy provides for any application is load balancing. For the BDD package, load balancing means that the BDD nodes are distributed evenly across the disks of a cluster. This is achieved by assigning each BDD node to a random disk. Hence, a BDD node is not necessarily stored near its children, and accessing the children of a node in an immediate manner would lead to long delays due to network and disk latency. Using delayed batched operations, as Roomy does (see Section 3), is the only acceptable solution if the very large BDDs are being treated as monolithic. Other approaches decompose the large BDD into smaller BDDs and then solve each of them separately on various compute nodes in the cluster [6]. These two solutions are not mutually exclusive. A decomposition approach can be adapted to our BDD package as well. For the rest of the paper, only the case of very large monolithic BDDs which are not decomposed into smaller BDDs is considered.

The parallel disk-based `apply` algorithm consists of two phases: an `expand` phase and a `reduce` phase, described in Algorithms 2 and 3, respectively. The `expand` phase creates a valid but larger than necessary QBDD. It can be reduced to a smaller size, because it contains non-unique nodes. Non-unique nodes are nodes at the same level i which represent identical sub-BDDs. However, the fact that they are duplicates cannot be determined in the `expand` phase. `expand` is a top-down breadth-first scan. The purpose of the `reduce` phase is to detect the duplicates at each level and keep only one representative of each duplication class. `reduce` is a bottom-up scan of the BDD returned by `expand`.

Notation.

The notation $k \rightarrow v$ is used to represent a key-value entry in a Roomy-hashtable.

All *id*s are Roomy-unique integers that can be passed back to Roomy as keys in a Roomy-hashtable.

A *forwarding node* is a temporary node created during the `reduce` phase. Such a node acts as a pointer to a terminal node, meaning that all parents pointing to the node should actually point to the terminal node. When it is no longer needed, it is removed.

If *node* is a forwarding node in the reduce phase, the notation $fwd(id_{node})$ is used to represent the id of the permanent node that *node* points to.

Implementation aspects.

The `expand` algorithm uses per-level distributed disk-based requirement queues Q and Q', which are implemented as Roomy-hashtables. All entries in Q are of the form $(id_A, id_B) \rightarrow (id_{AB}, parent\text{–}id_{AB})$.

The nodes at a certain level in AB are stored in $Nodes[i]$. In AB, $id_{AB} = op(id_A, id_B)$. id_{AB} is a low or high child of $parent\text{–}id_{AB}$, a node at the previous level in AB.

`expand` creates a partially-reduced OBDD, which `reduce` converts into a QBDD.

Note that our `expand` algorithm differs from the `expand` presented in [21] in the following way: in our case all data structures are distributed and stored on the parallel disks of a cluster; there is no need for any QBDD level to fit even in the distributed RAM at any time; all the operations that involve reading or writing remote data are delayed and batched; in line 7, unique ids for the left and right child of a node are created in advance (in the next iteration it will be found that some are duplicates), while in [21] a per-level memoization table named "operation-result-table" is used. When duplicates are found, the parent nodes of the duplicates are updated to point to the child representative of the duplicate class. A per-level memoization table would not work for parallel `expand` algorithms because all reads are delayed, and so one cannot immediately check whether a certain node has already been computed.

Algorithm 2 Parallel disk-based `expand`

Input: QBDDs A, B and Boolean operator op
Output: QBDD $AB = op(A, B)$

1: Initialize Q with the entry representing the root of AB: $(root\text{–}id_A, root\text{-}id_B) \rightarrow (new\ Id(), N/A)$
2: $level \leftarrow 0$, $Q' \leftarrow \emptyset$
3: **while** $level \leq max(depth(A), depth(B))$ AND $Q \neq \emptyset$ **do**
4: Remove entries with duplicate keys from Q and update their parent nodes (extracted from the entry's value) to point to the id found in the value of the representative of the duplicate class.
5: **for** each entry $(id_A, id_B) \rightarrow (id_{AB}, parent\text{–}id_{AB})$ in Q **do**
6: Perform delayed lookup of the child nodes of id_A : $low(id_A)$ and $high(id_A)$ and of id_B : $low(id_B)$ and $high(id_B)$
7: $id' \leftarrow new\ Id()$
$id'' \leftarrow new\ Id()$
Create two entries in Q':
$(low(id_A), low(id_B)) \rightarrow (id', id_{AB})$
and, respectively,
$(high(id_A), high(id_B)) \rightarrow (id'', id_{AB})$
8: Insert node $id_{AB} \rightarrow (id', id'')$ in $Nodes[level]$
9: $level \leftarrow level + 1$
10: $Q \leftarrow Q'$, $Q' \leftarrow \emptyset$

Algorithm 3 Parallel disk-based `reduce`

Input: a QBDD AB, returned by `expand`
Output: a QBDD AB', resulted from AB by eliminating non-unique nodes

1: $level \leftarrow depth(AB)$
2: **while** $level \geq 1$ **do**
3: **for** each node entry $(id) \rightarrow (low(id), high(id))$ in $Nodes[level]$ **do**
4: **if** $low(id)$ is a forwarding node **then**
5: $low(id) \leftarrow fwd(low(id))$
6: **if** $high(id)$ is a forwarding node **then**
7: $high(id) \leftarrow fwd(high(id))$
8: **if** $low(id) = high(id) = T$ (terminal node 0 or 1) **then**
9: Make this a forwarding node to T.
10: Remove non-unique nodes (duplicates) at the current level. Update the parents of the duplicate nodes to point to the representative of the duplicate class. This is duplicate detection.
11: $level \leftarrow level + 1$

Satisfying Assignments.

The `any-SAT` algorithm simply follows any path in the QBDD that ends in the terminal 1. When a high child is followed, the level's variable is set to 1. When a low child is followed, the level's variable is set to 0. When terminal 1 is reached, the assigned values of all the variables are listed.

The `SAT-count` algorithm is implemented with a top-down breadth-first scan of the QBDD. Each node in the QBDD has a counter associated with it. Initially, the root of the QBDD has its counter set to 1 and all other counters are set to 0. At each level in the breadth-first scan, the counter of each node is added to each of its children's counters. When a child of a node at level i is the terminal node 1, update a global counter by adding the node's local counter multiplied by 2^m, where $m = depth(QBDD) - i - 1$, to it. All updates are delayed and batched, to maintain the parallelism of the scan. After the scan finishes and the updates are processed, the global counter contains the number of satisfying assignments of the QBDD.

5. EXPERIMENTAL RESULTS

Hardware and Software Configurations.

The Roomy-based parallel disk BDD package was experimentally compared with BuDDy [8], a popular open-source BDD package written in C/C++. BuDDy was used with two different computer architectures.

- **Server:** with one dual-core 2.6 GHz AMD Opteron processor, 8 GB of RAM, running HP Linux XC 3.1, and compiling code with Intel ICC 11.0.
- **Large Shared Memory:** with four quad-core 1.8 GHz AMD Opteron processors, 128 GB of RAM, running Ubuntu SMP Linux 2.6.31-16-server, and compiling code with GCC 4.4.1.

While BuDDy could make use of all of the RAM on the machine with 8 GB, it was only able to use approximately 56 GB out of 128 GB in the other case, because the array used to store the nodes of the BDDs was limited to using a 32-bit index.

For Roomy, the architecture was a cluster of the 8 GB machines mentioned above, using a Quadrics QsNet II interconnect. Each machine had a 40 GB partition on the local disk for storage. This provides an upper bound for the total storage used at any time. For smaller problem instances, 8 machines of the cluster were used, with larger examples using 32 or 64 machines.

Test Problems.

Four cases were tested: BuDDy using a maximum of 1, 8, or 56 GB of RAM; and Roomy. These four cases were tested on two combinatorial problems:

- **N-queens:** counting the number of unique solutions to the N-queens problem.
- **3-dimensional tic-tac-toe:** counting the number of possible tying configurations when placing N X's and $64 - N$ O's in a 4×4×4 3D tic-tac-toe board.

In both cases, the experiments were run with increasing values of N. For N-queens, this increases the problem difficulty by increasing the dimension of the board. For 3D tic-tac-toe, this increases the difficulty by increasing the number of possible positions to consider (with maximum difficulty at 32 out of 64 positions for each X and O).

The rest of this section describes the details of each problem, how the solutions are represented as Boolean formulas, and gives experimental results.

5.1 Counting N-Queens Solutions

Problem Definition.

The N-queens problem considered here is: how many different ways can N queens be placed on an N×N chess board such that no two queens are attacking each other? The size

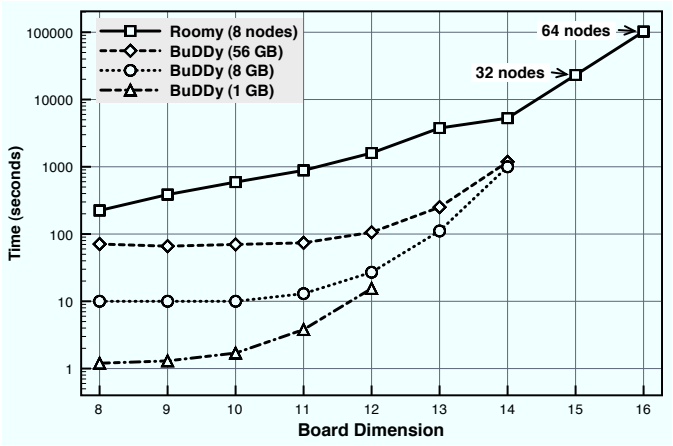


Figure 3: Running times for the N-queens problem.

of the state space is $N!$, corresponding to an algorithm that successively places one queen in each row, and for row k chooses one of the remaining $N-k$ columns not containing a queen.

The current record for the largest N solved is 26, achieved using massively-parallel, FPGA-based methods [26]. The more general BDD approach described here is not meant to directly compete with this result, but to serve as a method for comparing traditional RAM-based BDD packages and our parallel-disk based approach. The N-queens problem is often used as an illustration in introductions to the subject, and a solution to the problem comes with BuDDy as an example application.

The following defines a Boolean formula that represents all of the solutions to a given instance of the N-queens problem.

First, define N^2 variables of the form $x_{i,j}$, where $x_{i,j}$ is true iff a queen is placed on the square at row i column j. Then, define $S_{i,j}$, which is true iff there is a queen on square i,j and not on any square attacked from that position.

$$S_{i,j} = x_{i,j} \wedge \neg x_{i_1,j_1} \wedge \neg x_{i_2,j_2} \wedge \ldots$$

The constraint that row i must have exactly one queen is

$$R_i = S_{i,1} \vee S_{i,2} \vee \ldots \vee S_{i,N}$$

And finally, the board has one queen in each row.

$$B = R_1 \wedge R_2 \wedge \ldots \wedge R_N$$

The number of solutions is computed by counting the number of satisfying assignments of B.

Experimental Results.

Figure 3 shows the running times for the N-queens problem using BuDDy with a maximum of 1, 8, or 56 GB of RAM, and using our BDD package based on Roomy.

BuDDy with 1 GB is able to solve up to $N = 12$, which finished in 15 seconds. BuDDy with 8 GB is able to solve up to $N = 14$, which finished in 16.7 minutes. For higher N, BuDDy runs out of RAM. This demonstrates how quickly BDD computations can exceed available memory, and the need for methods providing additional space.

Even when given up to 56 GB, BuDDy can not complete $N = 15$. For smaller cases, the version using more memory is slower because the time for memory management dominates the relatively small computation time.

Table 1: Sizes of largest and final BDDs, and number of solutions, for the N-queens problem (* indicates that BuDDy exceeded 56 GB of RAM without completing).

N	Largest Buddy BDD	Largest Roomy BDD	Ratio
8	10705	11549	1.08
9	44110	50383	1.14
10	212596	234650	1.10
11	1027599	1105006	1.08
12	4938578	5250309	1.06
13	26724679	29370954	1.10
14	153283605	165030036	1.08
15	*	917859646	–
16	*	3380874259	–

N	Final Buddy BDD	Final Roomy BDD	Ratio	# of Solutions
8	2451	2451	1.00	92
9	9557	9557	1.00	352
10	25945	25945	1.00	724
11	94822	94822	1.00	2680
12	435170	435170	1.00	14200
13	2044394	2044394	1.00	73712
14	9572418	9572418	1.00	365596
15	*	51889029	–	2279184
16	*	292364273	–	14772512

The Roomy-based package is able to solve up to $N = 16$, which finished in 28.4 hours. The figure shows the running times for $N \leq 14$ using 8 compute nodes, $N = 15$ using 32 nodes, and $N = 16$ using 64 nodes. The results demonstrate that parallel-disk based methods can extend the use of BDDs to problem spaces several orders of magnitude larger than methods using RAM alone.

For the cases using BuDDy, the versions using more RAM had a time penalty due to the one-time cost of initializing the larger data structures. For Roomy, the time penalties for the smaller cases are primarily due to synchronization of the parallel processes, which is amortized for the longer running examples.

For the computations using Roomy, BuDDy was first used to compute each of the N row constraints, R_i. Then, Roomy was used to combine these into the final BDD B. This was done to avoid performing many operations on very small BDDs with Roomy, which would cause many small, inefficient disk operations.

Table 1 shows the sizes of the largest BDD, the final BDD, and the number of solutions for $8 \leq N \leq 16$. In this case, the largest BDD produced by Roomy was approximately 22 times larger than the largest BDD produced by BuDDy (for instances that were solved). Table 1 also shows that, for this problem, the additional nodes required by the use of quasi-reduced BDDs, instead of the traditional fully reduced BDDs, increase BDD size by at most 14 percent. We consider this cost acceptable given the increase in locality QBDDs provide the computation.

5.2 Counting Ties in 4×4×4 3D Tic-Tac-Toe

Problem Definition.

This problem deals with a generalization of the traditional game of tic-tac-toe: two players, X and O, take turns marking spaces in a 3×3 grid, attempting to place three marks in a given row, column, or diagonal. In this case, we consider

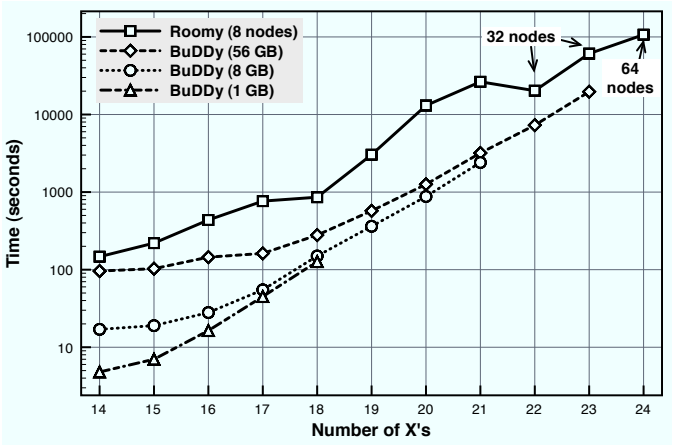


Figure 4: Running times for the 3D tic-tac-toe problem.

a 3-dimensional 4×4×4 grid.

The question considered here is: how many possible tie games are there when X marks N spaces, with O filling the remaining $64 - N$? As N is increased from 1 to 32, both the number of possible arrangements and the difficulty of the problem increase. To the best of our knowledge, this problem has not been solved before.

The Boolean formula representing the solution uses 64 variables of the form $x_{i,j,k}$, which are true iff the corresponding space is marked by X. First, a BDD representing placements that have exactly N out of 64 spaces marked with X is constructed. Then, BDD constraints for each possible sequence of four spaces in a row are successively added. The constraints are of the form: ¬(all four spaces are X) ∧ (at least one space has an X).

The number of possible ties is computed by counting the number of satisfying assignments of the final BDD.

Experimental Results.

Figure 4 shows the running times for the 3D tic-tac-toe problem using BuDDy with a maximum of 1, 8, or 56 GB of RAM, and using the Roomy-based package. As in the N-queens problem, BuDDy was used to compute several smaller BDDs, which were then combined into the final BDD using Roomy.

BuDDy with 1 GB of RAM is able to solve up to $N = 18$, finishing in 127 seconds. BuDDy with 8 GB of RAM is able to solve up to $N = 21$, finishing in 40 minutes. Unlike the N-queens problem, increasing the available RAM from 8 to 56 GB increases the number of cases that can be solved, up to $N = 23$, which finished in 5.5 hours. Roomy was able to solve up to $N = 24$, finishing in under 30 hours using 64 nodes.

Table 2 shows the sizes of the largest BDD, the final BDD, and the number of solutions for $14 \leq N \leq 24$. The smallest number of X's that need to be placed to force a tie is 20, yielding 304 tying arrangements. The number of possible ties then increases rapidly, up to over 5 billion for $N = 24$.

For the 3D tic-tac-toe problem, all of the QBDDs used by Roomy are exactly the same size as the ROBDDs used by BuDDy. So, like the N-queens problem, using the possibly less compact representation is not an issue here.

Table 2: Sizes of largest and final BDDs, and number of tie games, for the 3D tic-tac-toe problem (* indicates that BuDDy exceeded 56 GB of RAM without completing).

X's	Largest Buddy BDD	Largest Roomy BDD	Ratio
14	389251	389251	1.00
15	671000	671000	1.00
16	1350821	1350821	1.00
17	4378636	4378636	1.00
18	11619376	11619376	1.00
19	24742614	24742614	1.00
20	42985943	42985943	1.00
21	113026291	113026291	1.00
22	383658471	383658471	1.00
23	988619402	988619402	1.00
24	*	2003691139	–

X's	Final Buddy BDD	Final Roomy BDD	Ratio	# of Ties
14	1	1	1.00	0
15	1	1	1.00	0
16	1	1	1.00	0
17	1	1	1.00	0
18	1	1	1.00	0
19	1	1	1.00	0
20	8179	8179	1.00	304
21	433682	433682	1.00	136288
22	6560562	6560562	1.00	9734400
23	60063441	60063441	1.00	296106640
24	*	373236946	–	5000129244

6. CONCLUSIONS AND FUTURE WORK

This work provides a parallel disk-based BDD package whose effectiveness is demonstrated by solving large combinatorial problems. Those problems are typical for a broad class of mathematical problems whose large state space contains a significant degree of randomness.

Future work will tackle the verification of large industrial circuits. A parallel disk-based ROBDD representation will be provided, together with modified versions of the BFS algorithms that now use QBDDs. It is expected that this future implementation will perform better for industrial circuits, while we believe that QBDDs will still be more suitable to combinatorial problems. These expectations coincide with the motivation of using ROBDDs in [2] and [24].

Extensive research shows that, for many industrial problems, dynamic variable reordering can yield significant space savings, which get translated into time savings. Providing a parallel disk-based method for dynamic variable reordering is also part of the future work.

7. REFERENCES

[1] Y. an Chen, B. Yang, and R. E. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach. Technical report, 1997.

[2] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided design*, pages 622–627, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] F. Bianchi, F. Corno, M. Rebaudengo, M. S. Reorda, and R. Ansaloni. Boolean function manipulation on a parallel system using BDDs. In *HPCN Europe '97:*

Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, pages 916–928, London, UK, 1997. Springer-Verlag.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, New York, NY, USA, 1990. ACM.

[5] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] G. Cabodi, P. Camurati, and S. Quer. Improving the efficiency of BDD-based operators by means of partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(5):545–556, 1999.

[7] M. Carbin. Learning effective BDD variable orders for BDD-based program analysis, 2006.

[8] H. Cohen. BuDDy: A binary decision diagram library, 2004. `http://sourceforge.net/projects/buddy/`.

[9] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *In CHARME*, pages 129–145. Springer, 2005.

[10] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. A scalable parallel algorithm for reachability analysis of very large circuits. *Form. Methods Syst. Des.*, 21(3):317–338, 2002.

[11] S. Kimura and E. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. of IEEE International Conference onComputer Design: VLSI in Computers and Processors, 1990 (ICCD '90)*, pages 220 –223, sep 1990.

[12] D. Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. `http://roomy.sourceforge.net/`.

[13] D. Kunkle. Roomy: A system for space limited computations. In *Proc. of Parallel Symbolic Computation (PASCO '10)*. ACM Press, 2010.

[14] W. Mao and J. Wu. Application of wu's method to symbolic model checking. In *ISSAC '05: Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, pages 237–244, New York, NY, USA, 2005. ACM.

[15] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 501–507, London, UK, 1998. Springer-Verlag.

[16] S. Minato and S. Ishihara. Streaming BDD manipulation for large-scale combinatorial problems. In *DATE '01: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 702–707, Piscataway, NJ, USA, 2001. IEEE Press.

[17] H. Moeinzadeh, M. Mohammadi, H. Pazhoumand-dar, A. Mehrbakhsh, N. Kheibar, and N. Mozayani. Evolutionary-reduced ordered binary decision diagram. In *AMS '09: Proceedings of the 2009 Third Asia International Conference on Modelling & Simulation*, pages 142–145, Washington, DC, USA, 2009. IEEE Computer Society.

[18] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs — a compact, canonical and efficiently manipulable representation for Boolean functions. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pages 547–554, Washington, DC, USA, 1996. IEEE Computer Society.

[19] Z. Nevo and M. Farkash. Distributed dynamic BDD reordering. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 223–228, New York, NY, USA, 2006. ACM.

[20] Z. Nevo and M. Farkash. Distributed dynamic BDD reordering. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 223–228, New York, NY, USA, 2006. ACM.

[21] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided design*, pages 48–55, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[22] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstation. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 358–364, Washington, DC, USA, 1996. IEEE Computer Society.

[23] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill, and E. A. Emerson. Multi-threaded reachability. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 467–470, New York, NY, USA, 2005. ACM.

[24] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 635–640, New York, NY, USA, 1996. ACM.

[25] A. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: challenges and solutions. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 344, Washington, DC, USA, 1997. IEEE Computer Society.

[26] R. G. Spallek, T. B. Preußer, and B. Nägel. Queens@TUD, 2009. `http://queens.inf.tu-dresden.de/`.

[27] S. Stergios and J. Jawahar. Novel applications of a compact binary decision diagram library to important industrial problems. *Fujitsu scientific and technical journal*, 46(1):111–119, 2010.

[28] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *DAC '96: Proceedings of the 33rd Annual Design Automation Conference*, pages 641–644, New York, NY, USA, 1996. ACM.

[29] B. Yang and D. R. O'Hallaron. Parallel breadth-first BDD construction. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 145–156, New York, NY, USA, 1997. ACM.