PJBDD:
A BDD Library for Java and Multi-Threading

Dirk Beyer, Karlheinz Friedberger, and Stephan Holzner

LMU Munich, Germany

Abstract. PJBDD is a flexible and modular Java library for binary decision diagrams (BDD), which are a well-known data structure for performing efficient operations on compressed sets and relations. BDDs have practical applications in composing and analyzing boolean functions, e.g., for computer-aided verification. Despite its importance, there are only a few BDD libraries available. PJBDD is based on a slim object-oriented design, supports multi-threaded execution of the BDD operations (internal) as well as thread-safe access to the operations from applications (external). It provides automatic reference counting and garbage collection. The modular design of the library allows us to provide a uniform API for binary decision diagrams, zero-suppressed decision diagrams, and also chained decision diagrams. This paper includes a compact evaluation of PJBDD, to demonstrate that concurrent operations on large BDDs scale well and parallelize nicely on multi-core CPUs.

Keywords: BDD · Java Library · Concurrency · Multi-threaded Application

1 Introduction

Binary Decision Diagrams (BDDs) [1, 8] enabled a major break-through in applying model checking to large hardware models [9]. In our own previous work, we applied BDDs to model checking of timed automata [2] and C programs [5]. Most of the existing, state-of-the-art BDD libraries are not designed in thread-safe manner (CuDD [15], BuDDy [10], and JDD [16]), do not support multi-threaded execution of the BDD operations (BeeDeeDee [14]), or require effort to manually update reference counters for BDD nodes (Sylvan [11]). Therefore, application developers of, e. g., verification tools based on BDDs, have to implement code for cleaning up unused nodes or cannot directly use multi-threaded verification algorithms with BDDs in a thread-safe manner.

PJBDD contributes to closing this gap and offers a full-fledged BDD library with support for convenient usage from Java applications. Table 1 lists the programming and BDD features that we identified as important in our development work on the verification framework CPAchecker, which uses BDDs as a central data structure. For our work it is more important to have a convenient and thread-safe development environment with an *easy-to-read code basis*, than to

Table 1: Different BDD libraries and their features

	last maintained	thread-safe access	parallel operations	automatic reference counting	dynamic variable reordering	further supported diagrams
BuDDy [10]	2014	-	-	✓	✓	-
CuDD [15]	(2016)	-	-	-	✓	ADD, ZBDD, CBDD
PJBDD	2021	✓	✓	✓	-	ZBDD, CBDD
JDD [16]	2019	-	-	-	-	ZBDD
Sylvan [11]	2020	(✓)	✓	-	-	ADD, LDD, TBDD
BeeDeeDee [14]	2018	✓	-	-	-	-

leverage the maximal possible performance. This makes the library easier to maintain and extend for us and our students. PJBDD is also an interesting choice for teaching. PJBDD is the only available BDD library (Table 1) that

- is actively maintained by the developers,
- ensures thread-safe concurrent calls from user applications in Java,
- supports multi-threaded execution of BDD operations,
- provides automatic reference counting, and
- supports zero-suppressed BDDs (ZBDD) and chained BDDs (CBDD).

Related Work. BDDs are practically relevant since the seminal paper by Bryant in 1986 [7]. Several highly tuned BDD libraries became available since that time, written in different programming languages. Well-known examples are the C/C++ libraries BuDDy [10], CuDD [15], and Sylvan [11], as well as the Java libraries BeeDeeDee [14], and JDD [16].

The performance of a BDD library depends on several low-level design choices, which makes it difficult for researchers to develop new design approaches in existing highly optimized code. Furthermore, existing libraries often lack support for multi-threaded algorithms, concurrent access, or automatic reference counting. The Java-based implementation BeeDeeDee allows to perform thread-safe parallel operations. The library Sylvan [11] achieved great speed-up in large-scale scenarios with multi-threaded execution of BDD operations. However, due to a bug in the Java wrapper, thread-safe access from Java is not possible (An issue was reported at `https://github.com/utwente-fmt/jsylvan/issues/3`). While several tools support automated garbage collection, BuDDy and PJBDD are the only tools that support automated reference counting. The last date of official maintenance of CuDD is unknown, because the official FTP server is offline (The mirror at `https://github.com/ivmai/cudd` does not show activity since 2016.)

The implementation of ZBDDs is only available in the oldest (and thus most advanced) implementations; and unfortunately missing in newer libraries like BeeDeeDee and Sylvan. PJBDD closes this gap by providing all of the features described above in a well-known platform-independent programming language.

2 Design and Implementation Details

This section gives a compact overview of PJBDD's design and implementation.

Shared Graph Representation. In an application, there is not only one single BDD, but there are multiple of them. For overall efficiency, it is required that all common sub-graphs are shared in one central data structure, i.e., in a large hash table. In our BDD library, this shared data structure, called UniqueTable, is usable from multiple threads in concurrent manner, and we took care of minimal synchronization overhead. Therefore, read and write accesses are implemented as atomic compare-and-swap operations (CAS). Our hash tables use a prime-hashing function, which is a common choice for BDD libraries.

Operations Cache. For efficient BDD manipulation, a cache for computed operation results is necessary. Since the cache heavily reduces workload and achieves huge speed up, we implemented one central caching instance which all worker threads share. To enable thread-safe accesses we use atomic CAS accesses.

Concurrent BDD Operations. For concurrent operations, we use a fork-join parallelism. The Shannon expansion in the BDD applies its operations such that the two recursive calls run in parallel. We keep the implementation as simple as possible and use the Java-native fork-join framework to avoid overthreading and respect the execution order (the two recursive calls have to finish before returning).

Memory Management. Automatic memory management relieves the developer from the error-prone and tedious job of manually allocating and deallocating memory, which is one of the advantages of high-level programming languages like Java. An automatic garbage collection clears all memory objects that are no longer reachable from the user application. However, one problem remains that is crucial for long-running applications: The application can leak memory if the user forgets to remove object references in a central data structure. PJBDD offers automated cleaning of unused nodes. We chose to use *WeakReferences* and *ReferenceQueues* as provided by the JDK for fine-grained, but automatic and efficient memory control.

3 Architecture of the Library

Our library is written in Java. In comparison to other BDD libraries, PJBDD does not work with integer indices as internal BDD representations, but with Java objects. This allows us to use the object-oriented approach, but at the price of slightly heavier memory consumption. More implementation details and results of preliminary experiments are available in the Master's thesis by Holzner [13].

Design Criteria. Instead of developing another Java clone of an existing C library, we started from scratch and thoroughly considered the design criteria. Our development of a new BDD library is motivated by several requirements that are not addressed by existing BDD libraries.

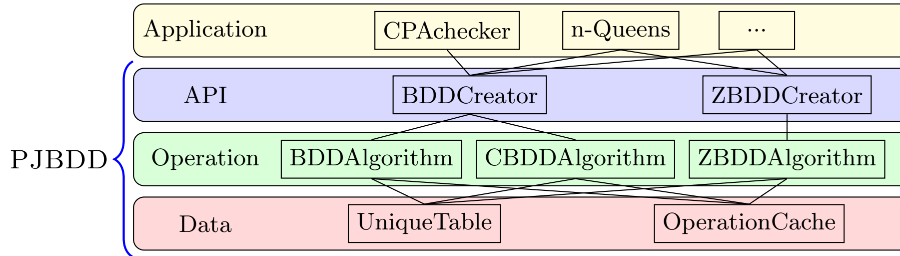

Fig. 1: Overview of the components of PJBDD

First, we desire a *simple to use API* and an *easy-to-read code basis*, such that future developers (including students) can experiment with and extend the existing code without requiring expert knowledge about optimizations, such as low-level bit-operations and reference counting. Of course, memory management is important for a highly optimized library. However, memory management by the user of a library is error prone and modern programming languages tend to already include automatic garbage collection. We decided for the standard Java garbage collector and do not provide an explicit way for the user to remove BDD nodes from the cache. Second, to minimize development time and maintenance costs, we used components from the Java standard API, such as the default fork-join framework for efficient multi-threaded computations.

Overview. Our library offers two distinct APIs: one for working with BDDs and one for ZBDDs. Due to their theoretical different nature, it is not possible to directly combine those types of decision diagrams. The API for BDDs provides typical boolean operations, such as conjunction, implication, or negation. Our library supports to configure chaining [6] with the same interface. The API for ZBDD has typical operations on ZBDDs, such as union and intersection.

Both APIs access the same kind of data structures: An operation layer, a node implementation, and a central cache. Figure 1 gives an overview of the layers and used components. The operation layer contains the basic algorithms on BDDs, and their implementation is optimized for multi-threaded computation. A BDD node itself represents an independent subtree and its implementation is as slim as possible to minimize memory consumption. A BDD node references its variable, the end of its chain in case of CBDD, and its two child nodes along the high and low edge. The central components of a BDD library are the node caches, which are divided into the global UniqueTable for node references given to the user, and the operation cache that is utilized in all internal algorithms. The operation cache is a crucial ingredient for BDD operations and responsible for the overall performance of the library.

With our modular approach, we can exchange several components to analyze the effect of different implementations without changing the user's application that is built on top of our library. For example, we can select from different cache and UniqueTable implementations or enable BDD chaining. For the experiment, we have set the currently best choices as default to evaluate the impact of concurrent computation on a scaling application.

Table 2: Solving the n-queens problem with a limited number of threads and a given number of CPU cores (wall time in seconds, memory consumption in MB)

	10-queens		11-queens		12-queens		13-queens	
cores	(s)	(MB)	(s)	(MB)	(s)	(MB)	(s)	(MB)
1	3.5	200	15	480	93	2 200	620	12 000
2	3.3	400	10	1 200	54	4 700	490	13 000
4	2.3	430	6.3	1 800	32	5 100	220	13 000
8	1.7	400	4.5	1 600	19	5 800	140	12 000

4 Experimental Evaluation

Our evaluation was executed with PJBDD, version v1.0.9 on an Intel Xeon E3-1230 CPU with 8 processing units. To guarantee reproducibility we isolated the benchmark runs with BenchExec [4], restricted the memory to 15 GB and set the maximal Java heap size to 12 GB. The *n-queens problem* is a typical satisfiability problem, which can be represented as BDD. To correctly solve the problem, one needs to place n chess queens on a chess board of size $n \times n$, such that no queen can be beaten by others (according to chess rules). BDDs can represent all the problem's different possible solutions in one BDD. To evaluate whether PJBDD scales well on multiple CPU cores, we analyzed the n-queens problem and measured the consumed memory and response time, when PJBDD uses a given number of CPU cores. The results in Table 2 show a significant impact of the parallelization. PJBDD's memory usage increased with multi-threaded computations (up to four times for $N = 11$). In terms of response time, our library can achieve a speed-up of up to five times for this application.

5 Conclusion

The abundance of multi-core environments makes it meaningful to invest in multi-threaded verification algorithms. This, however, requires the availability of thread-safe and multi-threaded data-structure libraries. The advent of Sylvan showed that this is possible and can lead to a considerable speed-up. Our motivation is to provide a Java implementation of a BDD library that guarantees thread-safe operation and supports multi-threaded execution of the BDD operations. PJBDD is such a BDD package. We use the n-queens problem as a load test and showed that the parallelization works well, as the work nicely distributes over the cores.

There are lots of additional features that can be implemented in the future. Due to the modular implementation, slim and flexible ZBDD and CBDD implementations are included already. This design could be used to support more different types of decision diagrams, such as CZBDDs [6] or tagged BDDs [12]. Improvements in performance or memory consumption without introducing additional code complexity is also a major goal of the developers.

Data Availability Statement. PJBDD is licensed under Apache 2.0 and available on GitLab: `https://gitlab.com/sosy-lab/software/paralleljbdd`. The

repository contains examples and instructions how to install and use the tool. A reproduction package for the n-queens experiment and some software-verification experiments is available on Zenodo [3].

Funding. This project was supported by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY)

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Computers **27**(6), 509–516 (1978). https://doi.org/10.1109/TC.1978.1675141
2. Beyer, D.: Improvements in BDD-based reachability analysis of timed automata. In: Proc. FME. pp. 318–343. LNCS 2021, Springer (2001). https://doi.org/10.1007/3-540-45251-6_18
3. Beyer, D., Friedberger, K., Holzner, S.: Reproduction package for article 'PJBDD: A BDD library for Java and multi-threading' in Proc. ATVA 2021. Zenodo (2021). https://doi.org/10.5281/zenodo.5070156
4. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
5. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS. pp. 1–11. LNCS 7721, Springer (2013). https://doi.org/10.1007/978-3-642-36046-6_1
6. Bryant, R.E.: Chain reduction for binary and zero-suppressed decision diagrams. J. Autom. Reasoning **64**(7), 1361–1391 (2020). https://doi.org/10.1007/s10817-020-09569-6
7. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
8. Bryant, R.E.: Binary decision diagrams. In: Handbook of Model Checking, pp. 191–217. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_7
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. LICS. pp. 428–439. IEEE (1990). https://doi.org/10.1109/LICS.1990.113767
10. Cohen, H., Whaley, J., Wildt, J., Gorogiannis, N.: BuDDy: A BDD package. `http://sourceforge.net/p/buddy/`
11. van Dijk, T.: Sylvan: Multi-core decision diagrams. Ph.D. thesis, University of Twente, Enschede, Netherlands (2016)
12. van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: Combining reduction rules from different decision diagram types. In: Proc. FMCAD. pp. 108–115. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102248
13. Holzner, S.: Design und Implementierung einer parallelen BDD-Bibliothek. Master's Thesis, LMU Munich, Software Systems Lab (2019)
14. Lovato, A., Macedonio, D., Spoto, F.: A thread-safe library for binary decision diagrams. In: Proc. SEFM. pp. 35–49. LNCS 8702, Springer (2014). https://doi.org/10.1007/978-3-319-10431-7_4
15. Somenzi, F.: Colorado University decision diagram package (1998)
16. Vahidi, A.: JDD: A pure Java BDD and Z-BDD library. `https://bitbucket.org/vahidi/jdd` (2003)