ParaQooba: A Fast and Flexible Framework for Parallel and Distributed QBF Solving*

Maximilian Heisinger[1](✉), Martina Seidl[1], and Armin Biere[2]

[1] JKU Linz, Linz, Austria, {maximilian.heisinger,martina.seidl}@jku.at
[2] ALU Freiburg, Freiburg, Germany, biere@informatik.uni-freiburg.de

Abstract. Over the last years, innovative parallel and distributed SAT solving techniques were presented that could impressively exploit the power of modern hardware and cloud systems. Two approaches were particularly successful: (1) search-space splitting in a Divide-and-Conquer (D&C) manner and (2) portfolio-based solving. The latter executes different solvers or configurations of solvers in parallel. For quantified Boolean formulas (QBFs), the extension of propositional logic with quantifiers, there is surprisingly little recent work in this direction compared to SAT. In this paper, we present ParaQooba, a novel framework for parallel and distributed QBF solving which combines D&C parallelization and distribution with portfolio-based solving. Our framework is designed in such a way that it can be easily extended and arbitrary sequential QBF solvers can be integrated out of the box, without any programming effort. We show how ParaQooba orchestrates the collaboration of different solvers for joint problem solving by performing an extensive evaluation on benchmarks from QBFEval'22, the most recent QBF competition.

1 Introduction

Quantified Boolean formulas (QBFs) extend propositional logic by quantifiers over the Boolean variables [2]. As a consequence, the decision problem of QBF (QSAT) is PSPACE complete, which is potentially harder than the NP-complete decision problem of propositional logic (SAT). Hence, the quantifiers allow for an efficient encoding of many reasoning problems from formal verification, synthesis, and planning [26] that most likely do not have a compact formulation in propositional logic. Over the last decade, considerable progress has been made in sequential QBF solving [22,21]. In contrast to SAT, where conflict-driven clause learning (CDCL) [19] is the predominant solving paradigm, in QBF solving different approaches of orthogonal strength have been presented. Besides QCDCL, the QBF variant of CDCL, which is implemented for example in the solver DepQBF [17], clausal abstraction as implemented in the solver Caqe [23] and abstraction-refinement based expansion as implemented in the solver RaReQs [13] are particularly successful [22,21]. All of these QBF solving approaches considerably benefit from preprocessing, i.e., an extra step before

* Supported by the LIT AI Lab funded by the State of Upper Austria.

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 426–447, 2023.
https://doi.org/10.1007/978-3-031-30823-9_22

the actual solving in which certain redundancies of a formula are eliminated in a satisfiability-preserving way with the aim to make it easier for the solver [10].

Despite the vivid development in sequential QBF solving, only few approaches have been presented for parallel and distributed QBF solving [18]. The most recent parallel QBF solvers are HordeQBF [1] which integrates sequential QCDCL-based solvers to obtain a parallel QBF solver and, more recently, a basic implementation of a QBF module based on the parallel SAT solver ParaCooba [6] with DepQBF as its only backend solver. To the best of our knowledge, besides these two approaches no other parallel QBF solver has recently been presented. The situation in SAT is different: several very powerful parallel and distributed SAT solvers like Mallob [24], Painless [5], and the afore mentioned solver ParaCooba [7] have been released. They show the potential of parallel and distributed approaches impressively by solving hard SAT instances, for example from multiplier verification [15].

In this paper, we present ParaQooba, a novel framework for parallel and distributed QBF solving that integrates search-space splitting based on the Divide-and-Conquer paradigm with portfolio solving. Our framework is built on top of the ParaCooba SAT solving framework and extends its basic non-portfolio QBF solving module. ParaQooba reuses most of ParaCooba's modules providing management and distribution of solver tasks. In addition, we implemented a very generic interface that allows the easy integration of any QBF solver binary into our framework.

Our main contributions are as follows:

- we present a new flexible framework for parallel and distributed QBF solving that combines D&C search-space splitting with portfolio solving;
- we show how different QBF solvers that are based on different solving approaches can be integrated seamlessly into our framework;
- we provide our framework as open-source project;
- we perform an extensive evaluation that demonstrates the power of our approach on various kinds of benchmarks.

ParaQooba is integrated into ParaCooba's and available on GitHub:

https://github.com/maximaximal/paracooba

This paper is structured as follows: First we introduce some preliminaries required for the rest of the paper in the following section. We continue with related work in section 3. After that, section 4 summarizes concepts of the ParaCooba solver framework used in our work. Then we introduce how we apply Divide-and-Conquer to solving QBF in section 5. Having introduced the background, we present our portfolio ParaQooba module in detail in section 6 and provide an extensive evaluation in section 7. Finally, we summarize our findings and conclude in section 8.

2 Preliminaries

We consider QBFs $\mathcal{Q}.\varphi$ in *prenex conjunctive normal form* (PCNF) where the *prefix* $\mathcal{Q}$ is of the form $Q_1x_1, \ldots, Q_nx_n$ with $Q \in \{\forall, \exists\}$. The *matrix* φ is a propositional formula over the variables $x_1, \ldots, x_n$ in conjunctive normal form (CNF). A formula in CNF is a conjunction ($\wedge$) of clauses. A *clause* is a disjunction ($\vee$) of literals. A literal is a variable x, a negated variable $\neg x$ or a (possibly negated) truth constant $\top$ (true) or $\bot$ (false). For a literal l, the expression $\bar{l}$ denotes x if $l = \neg x$ and it denotes $\neg x$ otherwise. We sometimes write a clause as a set of literals and a CNF formula as set of clauses. Further, it is often convenient to partition the quantifier prefix into *quantifier blocks*, i.e., maximal sets of consecutive sets of variables with the same quantifier type. For example, for the QBF $\forall x_1 \forall x_2 \exists y_1 \exists y_2.\varphi$ we also write $\forall X \exists Y.\varphi$ with $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$. With upper case letters $X, Y, \ldots$ (possibly subscripted), we usually denote sets of variables, while with lower case letters $x, y, \ldots$ (also possibly subscripted), we denote variables. If φ is CNF formula, then $\varphi_{x \leftarrow t}$ is the CNF formula obtained from φ by replacing all occurrences of variable x by truth constant $t \in \{\top, \bot\}$. Depending on the value of t, variable x is either set to true (if t is $\top$) or to false (if t is $\bot$). We define the semantics of QBFs as follows:

- a QBF $\forall X \mathcal{Q}.\varphi$ is true iff both QBFs $\forall X' \mathcal{Q}.\varphi_{x \leftarrow \bot}$ and $\forall X' \mathcal{Q}.\varphi_{x \leftarrow \top}$ are true where $x \in X$ and $X' = X \setminus \{x\}$;
- a QBF $\exists Y \mathcal{Q}.\varphi$ is true iff at least one of $\exists Y' \mathcal{Q}.\varphi_{y \leftarrow \bot}$ and $\exists Y' \mathcal{Q}.\varphi_{y \leftarrow \top}$ is true where $y \in Y$ and $Y' = Y \setminus \{y\}$.

Note that we assume that all variables of a QBF are quantified, i.e., we are considering closed formulas only. Further, we use standard semantics of conjunction, disjunction, negation, and truth constants. For example, the QBF $\phi_1 = \forall x \exists y.((x \vee y) \wedge (\neg x \vee \neg y))$ is true, while $\phi_2 = \exists y \forall x.((x \vee y) \wedge (\neg x \vee \neg y))$ is false. As we see already by this small example, the semantics impose an ordering on the variables w.r.t. the prefix. Given a QBF $\mathcal{Q}.\varphi$, we say that $x <_{\mathcal{Q}} y$ iff x occurs before y in the prefix. If clear from the context, we write $x < y$. In ϕ_1, we have $x < y$, while in ϕ_2, we have $y < x$.

3 Related Work

In practical QBF solving, attempts to parallelize and distribute QBF solvers have a long history (cf. [18] for a survey). Already more than 20 years back, the first distributed QBF solver PQSolve [4] was presented, in a time when QCDCL had not been invented yet. With the advent of QCDCL, several attempts have been made to build parallel QCDCL solvers and implement knowledge-sharing mechanisms for learned clauses and cubes. One example of such a solver is PAQuBE [16]. Unfortunately, the code of most of the early approaches is not available anymore. Following the success of Cube-and-Conquer-based search-space splitting, the QBF solver MPIDepQBF has been presented [14]. While MPIDepQBF does not implement any sophisticated look-ahead mechanisms,

it could demonstrate that even without knowledge-sharing considerable speedup could be achieved. These results serve as motivation for the approach presented in this paper. Unfortunately, MPIDepQBF is implemented in an older version of OCaml that does not run on recent systems and relies on now deprecated libraries, making a comparison impossible. As indicated by its name, it is tailored around the sequential QBF solver DepQBF [17]. Another recent MPI-based QBF solver is HordeQBF [1] which implements knowledge sharing for QCDCL solvers. It is designed in such a way that it allows the integration of any QCDCL solver. In order to integrate a solver, it requires that it implements a certain interface, i.e., programming effort is necessary to add a new solver. To the best of our knowledge, it includes the QBF solver DepQBF only. HordeQBF does not perform search-space splitting, but it is a parallel portfolio solver with clause- and cube sharing. It diversifies the parallel solver instances by different parameter settings. This is different than in sequential portfolio solvers as presented in [12], which select among different solvers based on some properties of the input formula. Overall, a very strong focus on QCDCL-based solvers can be observed for parallel QBF solving frameworks. Because of this, many chances for better solving performance are missed, as nowadays there are many other solvers of orthogonal strength. With ParaQooba we provide a simple way of exploiting the power of the different solving approaches without any integration effort.

4 ParaCooba

Our novel framework ParaQooba (with q in the middle of its name) builds on top of the SAT solver ParaCooba (with c in the middle of its name). In this section, we describe the parts of ParaCooba that are relevant for the remainder of this work for our extension of ParaCooba to ParaQooba.

ParaQooba will be made available publicly during the artifact evaluation under the MIT license, similar to ParaCooba [7,6] which is publicly available on GitHub also under the MIT license[3]. ParaCooba is a distributed Cube-and-Conquer (C&C) solver that implements a proprietary peer-to-peer based load balancing protocol. In contrast to standard D&C solvers the splitting of the search-space can both be done upfront by using a look-ahead solver that produces n cubes or online during solving by lookahead or other heuristics. Amongst other information, the cubes are stored in a binary tree, the *solve tree*.

Solver module. A *solver module* manages the sequential solver that is responsible for solving a subproblem. Different solver modules have different code-bases, but they also generally share common concepts. A solver module implements a parser task, which is created directly after the module was initiated and serves as its starting point. It parses the input formula in its own worker thread and instantiates a solver manager based on the fully parsed formula. The parser task also creates the first solver task as the root of the solve tree.

[3] github.com/maximaximal/Paracooba

Solver Tasks. For ParaCooba, *solver tasks* are paths in the solve tree, whith a *parser task* being used to generate the tree's root. Solver tasks are usually started as children of other tasks, saving references to their parents, with the root solver task being the only exception. A task's depth in the solve tree represents its priority to be worked on: The greater the depth, the more important a task is to be solved locally and the less important it is to be offloaded to other compute nodes by the broker module. Only tasks that were created locally may be distributed.

Broker module. The *broker module* handles relations between solver tasks and processes their results. While the solver module generates tasks, the broker schedules them based on their priorities (their depths) and offloads them if a different compute node has less load than the current node. A task result is propagated upwards across compute nodes, there is no conceptual difference between locally and remotely solved tasks. The broker module is generic and does not rely on a specific solver module, instead providing the environment a solver module works in. It is already provided by ParaCooba and stays the same for different solver modules.

Cube Sources. For generating concrete subproblems, *cube sources* provide assumption literals to leaf solver tasks. A cube source decides whether a given solver task should split again, based on the current configuration (mainly the splitting depth) and the given formula. Every solver module can implement its own cube source, hence there are different kinds of cube sources for different solver modules. On this basis, very flexible mechanisms for the selection of splitting variables can be implemented, ranging from a simple count of literal occurrences to advanced look-ahead heuristics.

Task Tree. The *task tree* built lazily, i.e., only once a leaf is visited, the leaf is either expanded into a sub-tree, or solved. We picture such a tree in Figure 1. This tree has a *depth* of 1, because the path from the tree's root solver task to the leaf solver tasks has a length of 1. Once the active cube source stops further splits from being carried out, the tree's maximum depth is reached. The worker thread currently executing a task then lends a solver instance from the solver manager's central store. Each solver instance is created on-the-fly once (normally initialized based on the parser task) for each worker thread, which can also happen for multiple worker threads in parallel. After a solver instance was created, all other tasks solved by the same worker thread use the same solver instance.

Guiding Paths. The cubes that are given to solver instances as assumptions are called *guiding paths*. They are generated from the path to the leaf being solved. The solver instance then handles the solving internally, blocking the worker thread until either result is generated or the task is terminated. Results are not returned to parents, but instead handled by the broker module, which then traverses the solve tree upwards as far as possible, based on the results already in

the tree. Different kinds of evaluations can be defined on every level using a user-defined *assessment function*. With the result processed by the broker module, the solver task then finishes and the worker thread can take on the next task, based on the next-highest priority. The broker may delete the solver task after it finished processing, if the result was already used somewhere above it in the tree and no information from the original solver task structure is required anymore. Once the broker module has enough information to solve the root task, the result of the formula was computed successfully.

Solver Handle. A *solver handle* wraps instances of a given solver. It must be able to receive an *Assume* event, directly followed by a *Solve* event. While processing these events, a correctly working handle must block its calling thread until a result is found. Additionally, it must be fully re-entrant after finishing processing, so that the next solver task can apply new assumptions. On top of this, a handle must also be able to process a *Terminate* event, stopping the solver and early-returning control to its calling thread. Such a termination event may happen at any time, as it is generated by other solver tasks. This possibility of random terminations was an issue for our extension to ParaQooba, as it complicated synchronization of all involved threads.

QBF Solver Module. ParaCooba already provided a basic *QBF solver module* similar to the approach seen in MPIDepQBF. It implemented a QDIMACS-parser in a new solver module based on the SAT module. It realizes a simple cube source that returns the variable at the nth position in the prefix, with n being the current depth of a solver task. The solve tree is built using two adapted assessment functions: one for variables quantified $\forall$ (requiring all sub-trees to be true), one for $\exists$ (requiring at least one sub-tree to be true). The assessment functions also use ParaCooba's cancellation-support to terminate unneeded siblings after results already satisfy the respective subproblem. As backend solver, it exclusively uses DepQBF that provides an incremental API (which no other recent solver provides, to the best of our knowledge).

Summary. With its already existing tree-based QBF solving module together with its support for distributed solving, ParaCooba provides a stable basis for building an advanced parallel QBF solver. While the existing QBF module is rather uncompetitive with a few exceptions that indicate its potential, its core infrastructure turned out to be very useful to build our novel framework ParaQooba that offers built-in portfolio support.

The networking support mentioned above enables combining multiple compute nodes by giving each peer a connection to the main node. This is achieved with setting the `--known-remote` option. With this feature it becomes possible to easily distribute larger problem instances on a cluster or in the cloud.

5 Architecture of ParaQooba: Combining Divide-and-Conquer Portfolio Solving

Our framework ParaQooba combines Divide-and-Conquer (D&C) search space splitting with portfolio solving. The key feature of ParaQooba compared to ParaCooba is to allow portfolio solving at different search depths. The idea is illustrated in Figure 1. Both approaches are widely used to realize parallel and distributed SAT and QBF solvers. The D&C approach has been especially successful for hard combinatorial SAT problems [11] in a variant called Cube-and-Conquer (C&C). The C&C approach relies on powerful, but expensive lookahead solvers that heuristically decide which variables shall be considered for splitting. In its original SAT version, ParaCooba builds upon this idea [7].

For a QBF $Q_1XQ_2Y\mathcal{Q}.\varphi$ with $Q_1 \neq Q_2$ and $Q_1, Q_2 \in \{\forall, \exists\}$ though, the possible choices for variable selection are more restricted because of the quantifier prefix. In general, only variables from the outermost quantifier block Q_1X may be considered, because otherwise, the value of the formula might change. Jordan et al. [14] observed that for QBF following the sequential order of the variables in the first quantifier block already leads to improvements compared to the sequential implementation of DepQBF. The already existing QBF solver module of ParaCooba (see section 4) relied on this observation: it traverses the prefix of a PCNF and splits each visited leaf into two sub-trees, respecting both universal and existential quantifiers, until a pre-defined maximum depth is reached. Hence, it re-implements the approach of MPIDepQBF in ParaCooba.

Our framework ParaQooba generalizes the previous QBF module of ParaCooba not only by generalizing the interface in such a manner that any QBF solver can be easily (without programming effort) integrated as backend solver. Now it is also possible to run several solvers in the leaves as shown in Figure 2 for one split. Overall, ParaQooba realizes the following approach. The search-space is split according to the variable ordering of the prefix until a given depth. Once one of the sub-trees of an existentially quantified variable split is found to be true, the other sibling is terminated. Only when both siblings return false, the whole split returns false. Universal splits work in a dual manner: the result is only true if both sub-trees are found to be true and false otherwise. This property of QBF enables efficient termination of sub-tasks.

In ParaQooba, we now also parallelize each solver call over several QBF solvers with orthogonal strategies. Compared to prior approaches [18], we run a portfolio of multiple solvers in the leaves of the solve tree instead of only parallelizing its root. Having just one tree leads to several advantages: We are more flexible and may also call a preprocessor (e.g. Bloqqer) before each solve call. We also only instantiate the tree once, saving memory and enabling early-termination of sibling solver tasks.

6 Implementation

This section describes the extension of the SAT solver ParaCooba (for an overview see section 4) to our QBF solving framework ParaQooba. As Para-

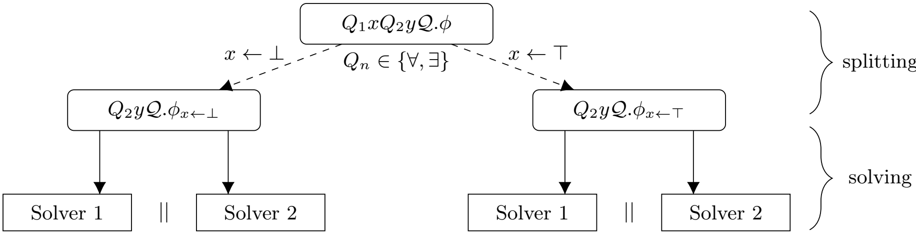


Fig. 1: Divide-and-Conquer with arbitrary-many levels of splitting and subformulas on the leaves solved by a portfolio of different sequential solvers

Cooba was originally not designed for portfolio support, several modifications and extensions were necessary. To this end, we first present the new QBF module of ParaQooba followed by a discussion of novel search-space pruning facilities.

6.1 The ParaQooba QBF Module

We generalized the already existing QBF solver handle to become an abstract base class, which now can be either a single solver handle or a *portfolio handle*. The latter unifies multiple handles into one, emulating a blocking and re-entrant interface. Once a portfolio handle is initialized, it starts one thread per internally wrapped handle. Each such thread implements a small state machine, waiting for events on a shared queue. Once the portfolio handle receives an assumption (a temporary truth assignment of a variable for one solver call), it is forwarded to all internal threads and is worked on by each wrapped solver in parallel.

If a portfolio handle was terminated before a solve call was issued, the internal handles would enter an invalid state. To circumvent this situation, an assumption event also directly triggers the internal state machine to continue into the solve state. Once the solve request actually arrives, it is just translated to an empty event, which, after it finished processing, indicates that a result was computed. A termination event is forwarded to the internal solver handles, but is limited to only one event per solve cycle.

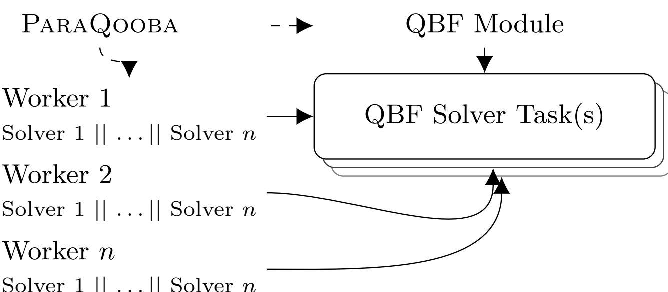


Fig. 2: The ParaQooba framework

The first internal solver handle to compute a result returns and sends a termination event to all sibling solvers. The result is saved and the portfolio handle waits for all internal handles to be ready to receive the next assumption, i.e., returning all solvers to a known state. Once every internal handle has reached that, the portfolio handle finally returns to its calling thread, forwarding the result of the inner handle. Because of thread scheduling and fast solving of trivial subproblems, a result can be forwarded even before the other sibling has been started, letting the broker module already complete a task before it itself has created both child tasks. This effect lead to some issues and had to be mitigated by adding some conditions on a task already being terminated even though it did not yet run to completion. Because a task will only be scheduled after the initial call to its assessment function, not many such checks were needed.

As many QBF solvers lack APIs, we have to work with their binaries that generally only read QDIMACS files. For this, we use the QuAPI interfacing library, that adds well-performing assumption-based reasoning support to generic solver binaries [9]. By not relying on specialized modifications of a solver's source code, we are able to plug-in generic third-party solvers, completely composable at runtime. Our ParaQooba module provides the `--quapisolver` parameter, that either directly specifies the leaf solver to be used, or automatically generates a portfolio handle to wrap multiple parallel leaf solvers. Note that our approach works for QBFs starting with existential as well as with universal quantification.

In its standard configuration, ParaQooba returns whether a given instance is found to be true or false. When enabling trace output using `-t`, it also supports printing the specific solver and the subproblem (including its guiding path) that produced a result. Using this machinery, one obtains an environment to experiment with benchmarks and to see how multiple solvers complement each other for the generated sub-formulas. The trace output is also useful when fully expanding a QBF formula by specifying a tree-depth of -1. While not advised for any real formulas, this was a well-received debugging aid for stress-testing new features. The opposite to this can also be done, by applying a tree-depth of 0. This directly solves the root task, without splitting the formula. This was also how the configuration PQ Portfolio with depth 0 (as discussed in the experimental evaluation below) was executed.

6.2 Search-Space Pruning

Preprocessing in the leaves. We modified the QBF preprocessor Bloqqer to allow forwarding output directly into a given solver binary by adding a `-p` argument. Internally, this writes the complete formula with added assumptions into the standard input of Bloqqer's preprocessing pipeline.

To plug e.g. Caqe into such a processing chain and then into ParaQooba, one may use our QBF solver module's command line option `--quapisolver bloqqer-popen@-p=caqe`. Deferring preprocessing until solving the leaves preserves the original formula structure of a formula during the split phase. We discuss the effects of this later in subsection 7.4.

Integer-Split Reduction. In many planning and verification encodings, the variables of a quantifier block QX are interpreted as bitvectors representing m nodes of a graph. Assume that $n = |X|$ bits with $m \leq 2^n$ are used for modeling the states of the graph. Then $2^n - m$ assignments to X are not relevant, but as a solver is agnostic of this information, it has to consider all assignments.

If m is known to the user, ParaQooba can be called with the option `--intsplit` (once or multiple times, once for each layer). One integer-split is counted as one layer in the task tree, so a tree-depth of two would split another quantifier into two more tasks for each state encoded in the previous integer-based split. To provide an example: Setting `--intsplit 5` creates 5 child-tasks in the task tree, spanning over the first $\lceil \log_2 5 \rceil = 3$ boolean variables from the quantifier prefix. When not using doing an integer-based split, these 3 variables would have to be expanded over 3 layers in the task tree, each inner task being split into two child tasks, resulting in 8 leaves , opposed to the 5 from before. Thus, integer-based splits require less intermediate splitting tasks to model the same formula, reducing the work to be done by the load-balancing mechanism in the Broker module. These integer splits are efficiently distributed over the network by relying on both the config-system and an extended QBF cube source. The cube source always saves the current guiding path, applying new splits, and in turn new assumptions, by appending to that path. The cube source itself is automatically serialized when a task is chosen to be offloaded to another compute node. While the possible savings are large, one has to exert great caution when using this feature, as it might change the semantics of a formula.

7 Evaluation

In this section, we evaluate ParaQooba on recent benchmarks and compare it to (sequential) state-of-the-art QBF solvers. As sequential backend solvers, we use the latest versions of DepQBF [17] as QCDCL solver, Caqe [23] as clausal-abstraction solver, and RaReQs [13] as recursive abstraction refinement solver. For preprocessing, we use Bloqqer [3] (version 31). All of these solvers were top-ranked in the most recent edition of QBFEval'22 [22]. For our experiments we used the benchmarks of the PCNF-track of this competition. The main questions we want to answer with our evaluation are as follows:

- how does the parallel portfolio-leaf approach of ParaQooba perform in comparison to the individual sequential solvers?
- how does the parallel portfolio-leaf approach of ParaQooba perform in comparison to the virtual portfolio solver of the sequential solvers?
- what is the impact of performing the preprocessing in the leaves instead on the original input formula?

We ran our experiments on machines with dual-socket 16 core AMD EPYC 7313 processors with 3.7 GHz sustained boost clock speed and 256 GB main memory. Each task was assigned as many physical cores as its setup required, except for tasks with more than 32 concurrent threads, which were exclusively

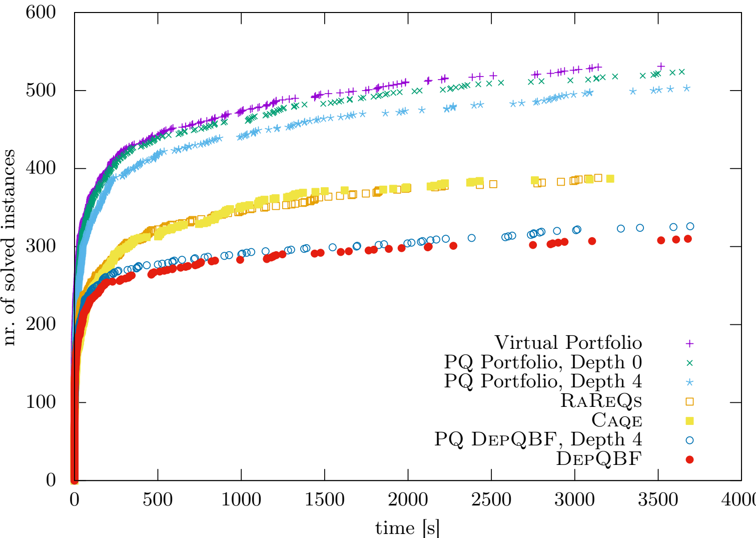


Fig. 3: Full summary of all solved instances with all different solvers without preprocessing. While Divide-and-Conquer (Depth 4) formulas solves 33 instances that no sequential solver solved, it solves 28 instances less in total.

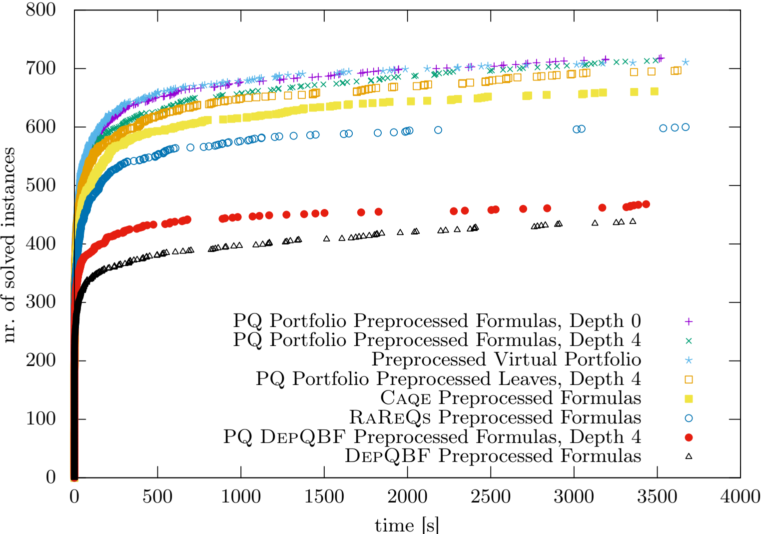


Fig. 4: Full summary of all solved instances with all different solvers with Bloqqer preprocessing. PQ Portfolio (Depth 4) solves 45 instances no sequential solver could solve and solves 3 more in total.

assigned a whole node each as to not be slowed down by other loads. The effects of over-committing in case of three concurrent portfolio solvers (48 threads running in parallel with only 32 physical cores available) are discussed below in subsection 7.3.

Please note that in this evaluation we do not use the networking features provided by ParaCooba, as we focus on applicability to QBF and not on the already presented scalability of the networking component (for the details see [3]).

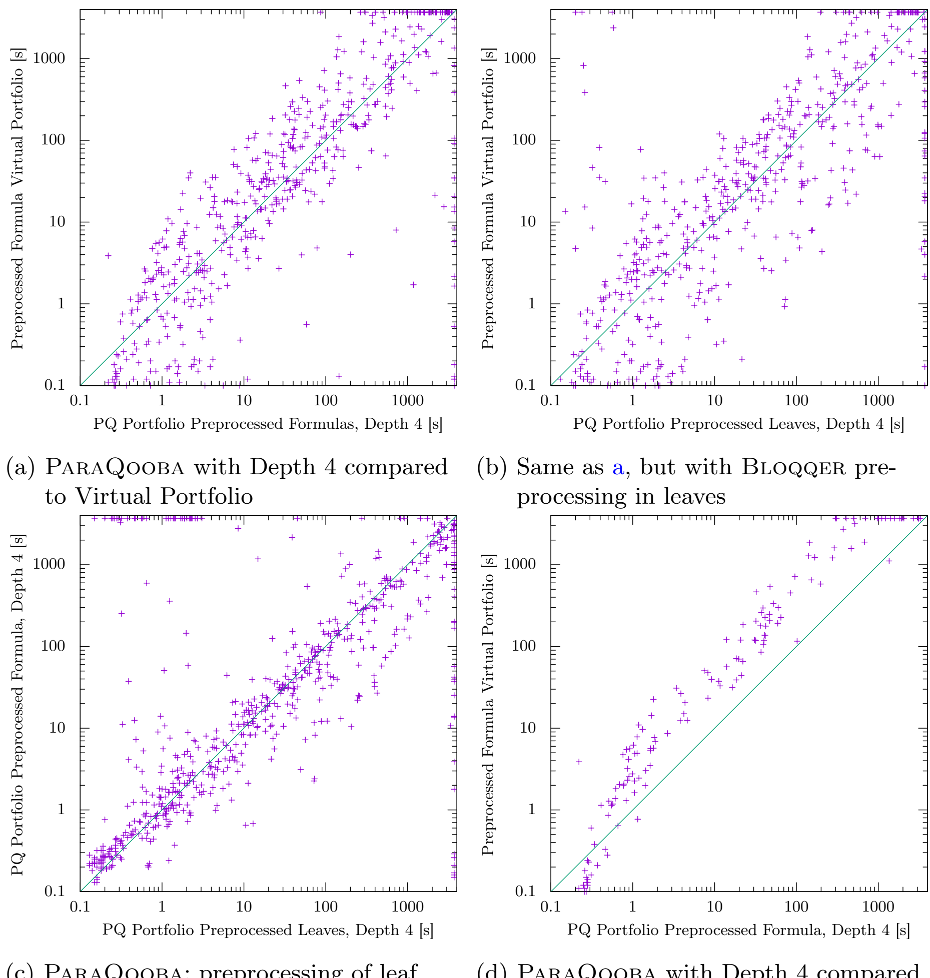
7.1 Overall Performance Comparison

In order to exploit our hardware with 32 physical cores and 64 logical cores in the best possible way, we mainly focus on a *splitting depth of four* in the following. With this depth, 16 worker threads are generated for each problem and with three sequential backend solvers, overall 48 processes are started. We call this configuration *PQ Portfolio, Depth 4*. For understanding the impact of splitting, we also consider other depths as well. With *PQ Portfolio, Depth 0* we refer to the configuration in which splitting is disabled. This configuration is particularly interesting, because compared to the virtual best solver (VBS), it reveals the overhead introduced by our framework (see also the discussion below). In order to show the improvements of ParaQooba compared to the QBF module without portfolio solving that was already available in ParaCooba [6], we also included the configuration *PQ DepQBF, Depth 4*.

Figure 3 shows the overall results of our evaluation *without preprocessing*. Both configurations of ParaQooba, *PQ Portfolio, Depth 0* and *PQ Portfolio, Depth 4* are considerably better than the single sequential solvers as well as the basic non-portfolio QBF module of ParaCooba only solving with DepQBF (PQ DepQBF, Depth 4). However, compared to the virtual portfolio, 28 instances less are solved in total (for an explanation see below). On the positive side, 33 formulas can be solved by our new approach that could not be solved by any sequential solver. The situation changes when preprocessing is applied (cf. Figure 4). Now ParaQooba in configuration *PQ Portfolio Preprocessed Formulas, Depth 4* is able to solve most formulas. It even solves more formulas than the *Preprocessed Virtual Portfolio*, indicating the potential of our approach.

A detailed analysis is given in Figure 5. By comparing the number of solved instances to the solve time of individual (preprocessed) problem instances, we see a small average speedup when using ParaQooba with depth 4 compared to a virtual portfolio solver in Figure 5a. The more trivial instances tend to be solved quicker using a sequential solver, while the harder to solve instances tend to be solved faster with the Divide-and-Conquer approach of ParaQooba.

Next, we used the preprocessed leaves functionality introduced in subsection 6.2. Here ParaQooba generates its guiding paths using the original formula and applies Bloqqer only in the leaves of the solve tree. In this configuration, some problem instances take longer to solve than when preprocessing the full formula, while others can be solved quicker. We present these results in Figure 5b. Such a result was expected, as it is conceptually similar to inprocessing.



(a) ParaQooba with Depth 4 compared to Virtual Portfolio

(b) Same as a, but with Bloqqer preprocessing in leaves

(c) ParaQooba: preprocessing of leaf formulas compared to preprocessing of input formula

(d) ParaQooba with Depth 4 compared to Virtual Portfolio on Hex formulas

Fig. 5: Detailed comparison of ParaQooba against the virtual portfolio of DepQBF, Caqe, and RaReQs in a, b, d. In a, ParaQooba solves 45 instances that no sequential solver could solve. In b, ParaQooba solves 38 instances no sequential solver could solve, 8 of which also could not be solved with portfolio over preprocessed formulas as in a. d focuses only on preprocessed formulas from the Hex benchmark family. In c, we directly compare preprocessing in the leaves to preprocessing in the input formula.

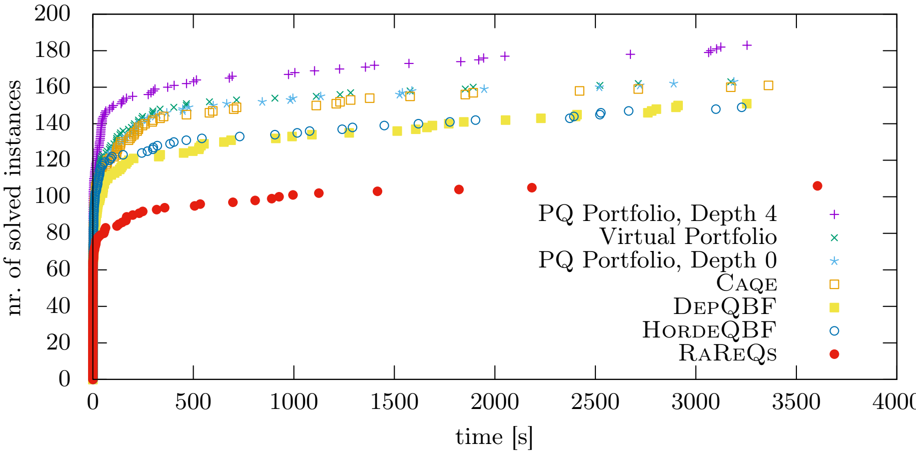


Fig. 6: Preprocessed formulas of the Hex positional game planning [20,25] benchmarks from the QBF22 benchmark set. Also compared to HordeQBF [1] as available state-of-the-art parallel QBF solver.

When considering the formulas that were exclusively solved by ParaQooba, then the variant with preprocessing the full formula up-front performed best followed by the variant with preprocessing in the leaves. These formulas include verification and synthesis benchmarks with 2–3 quantifier alternations as well as many encodings of the game Hex with 13, 15 or 17 quantifier alternations. Table 1 in the appendix lists all instances (48) that were only solved with some variant of ParaQooba. It also lists which variant was the fastest.

7.2 Family-Based Analysis

To understand which formula families benefit most from our Divide-and-Conquer solving strategy, we compared the (wall-clock) solve time of ParaQooba to the virtual portfolio solver. We calculated the speedup by dividing the solve time of the sequential solver by the solve time of ParaQooba. The instances with the highest speedups were some reachability queries (up to 18.09), the Hex game planning family (17.64), multipliers (16.46), and the formula_add family (15.16). More detailed results are appended in Table 2. Together with the number of Hex instances only ParaQooba solved (21), this makes Hex game planning the benchmark family with the best overall results in our evaluation. A comparison between ParaQooba and other solvers is shown in Figure 6.

7.3 Scalability of our Approach

As already discussed above, using 16 workers leads to overcommitting cores when solving with a portfolio of more than two solvers. To quantify this, we did

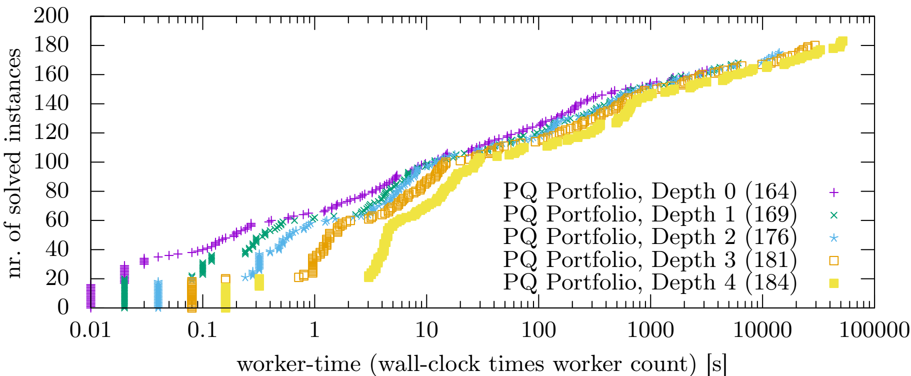


Fig. 7: Hex Scalability with preprocessed formulas. Depth 4 suffers from over-committing the available CPU-cores on our hardware and is relatively slow for the first few problems, but still solves more instances overall.

a scalability experiment with different worker counts. Because the Hex planning benchmarks had the most predictable performance, we focused this experiment on these formulas. Figure 7 shows the scalability graph, where the X-axis has been multiplied by the number of workers used, to visualize the cost of increased CPU-time compared to reduced wall-clock solve time. The impact of over-committing CPU cores can be clearly observed in the results of the portfolio with depth 4. This curve solves more compared to the others and takes longer to solve the first 140 instances, until the curves become more similar again.

7.4 Preprocessed Leaves compared to Preprocessed Formulas

We compared preprocessing the whole formula at once using Bloqqer to calling Bloqqer using `bloqqer-popen` in each leaf after first splitting on the unchanged formula. The first variant modifies the original prefix, including the quantifier ordering. Because the used splitting algorithm generates guiding paths by following this quantifier ordering, the different approaches lead to vastly different results. Figure 5c visualizes these differences by scattering both variants together.

Looking at the specific benchmarks benefiting from the two variants, we often observed improvements to one variant per family. This strongly suggests that adaptive preprocessing and inprocessing techniques could further improve solving performance, even without otherwise changing solvers themselves.

7.5 Lessons Learned

One would expect that for any given problem, parallel portfolio solvers are as fast as the fastest used solver. While this statement is conceptually true, we encountered some formulas where PQ-Portfolio gave comparatively bad results, while a solver alone could solve the same formula quicker or even instantly.

We investigated this in more detail and found several segmentation faults in Caqe and API inconsistencies in DepQBF that were encountered because of some corner-case structures of the generated subproblems (e.g., by enforcing the values of certain variables). We reported these issues to the solver developers and hope to obtain fixes soon. Having this issues fixed would lead to a more performant general solution and to a more robust user experience. In sequential execution of these solvers, we did not encounter any problems on the unmodified competition benchmarks without added unit clauses.

Currently, we adopt the following work-around. Segmentation faults of the sequential solvers are handled in our QBF module using the indirection provided by QuAPI. Once an unrecoverable error occurs in the solver child process, it exits and returns the error up through QuAPI's factory process and into the solver handle. There, such a result is interpreted as *Unknown*, which is invalid and therefore ignored, letting the portfolio wait for other results. We provide all affected formulas that we found in the artifact submitted alongside this paper.

We also observed that calling a solver via its API might lead to a considerably different behavior than calling a solver from the command line, i.e., different optimizations are activated when calling a solver through its API compared to using the command-line binary. Such behavior can be mitigated by not using the API directly, and instead relying on QuAPI, even if an API would be available. This fixes the issues with DepQBF, which solves some formulas (with assumptions supplied as unit clauses) in under one second if used as a solver binary, but not when applying assumptions through its API. We also supply all found formulas that triggered this issue in the submitted artifact.

8 Conclusions

We presented ParaQooba, a parallel and distributed QBF solving framework that combines search-space splitting with portfolio solving. We designed the framework in such a way that any sequential QBF solver binary can be easily integrated without any implementation effort. Our experiments demonstrate that this approach in combination with sequential preprocessing lead to considerable performance improvements for certain formula families.

With our framework, we provide a stable infrastructure that has the potential for many future extensions. For example, we did not incorporate any advanced splitting heuristics as in modern Cube-and-Conquer solvers. We expect that with more advanced heuristics, combined with adaptive but possibly non-deterministic re-splitting of leaves, even more speedups could be achieved.

In addition to the presented experiments, we also evaluated the novel integer-split feature (cf. subsection 6.2) with the Hex benchmark family. By providing the number of valid game states to ParaQooba, we could increase the splitting depth as well as the number of solved instances. We see much potential of providing encoding-specific or domain-specific knowledge to the solver and will investigate this in future work.

Data Availability Statement

Data used for benchmarking the described software, including source code, are made available permanently under a permissive license in a public artifact on Zenodo. Raw source data for the figures presented in this paper are also included [8].

A Instances Only Solved by ParaQooba

Name	Clauses	Variables	QA	Time [s]	Res	Variant
b21_C_3_206	242896	3270	3	265.77	⊤	full
c1_Debug_s3_f1_e1_v1	1775758	379113	3	3164.34	⊤	full
c2_Debug_s3_f1_e1_v2	431970	98425	3	1834.27	⊤	full
cache-coherence-2-fixpoint-2	10648	3686	2	0.56	⊥	leaves
cmu.dme1.B-f3	4540	1795	3	0.2	⊤	leaves
cmu.dme2.B-f3	6151	2342	3	818.3	⊤	leaves
LoginService	21667	5289	2	1086.07	⊥	orig
query64_query42_1344n	3423	1426	2	86.73	⊤	full
hex_compact_goal_witness_based_hein_03_6x6-13.pg	3401	1056	15	2594.27	⊥	leaves
hex_compact_goal_witness_based_hein_05_6x6-13.pg	3493	1071	15	3102.97	⊤	full
hex_compact_goal_witness_based_hein_17_6x6-13.pg	3430	1060	15	1919.64	⊥	full
hex_compact_goal_witness_based_hein_18_7x7-13.pg	4256	1267	15	1401.12	⊥	full
hex_compact_goal_witness_based_hein_02_5x5-13.pg	3134	1007	15	308.99	⊤	full
hex_compact_goal_witness_based_hein_15_5x5-15.pg	3667	1195	17	3063.67	⊤	full
hex_symbolic_explicit_goal_hein_03_6x6-11.pg	3421	902	13	693.11	⊥	full
hex_symbolic_explicit_goal_hein_05_6x6-11.pg	3611	918	13	501.29	⊥	full
hex_symbolic_explicit_goal_hein_18_7x7-11.pg	3084	1021	13	447.7	⊥	leaves
hex_symbolic_explicit_goal_hein_02_5x5-11.pg	2480	739	13	973.33	⊥	full
hex_symbolic_explicit_goal_hein_16_5x5-11.pg	2376	731	13	301.31	⊥	full
hex_symbolic_implicit_goal_hein_03_6x6-13.pg	3069	1001	15	1830.57	⊥	full
hex_symbolic_implicit_goal_hein_17_6x6-13.pg	3097	1005	15	2674.38	⊥	full

hex_symbolic_implicit_goal_hein_02_5x5-13.pg	2812	952	15	404.36	⊤	full
hex_symbolic_implicit_goal_hein_15_5x5-15.pg	3106	1072	17	1944.27	⊤	full
hex_witness_based_hein_03_6x6-13.pg	7174	1917	13	2050.04	⊥	full
hex_witness_based_hein_05_6x6-13.pg	7456	1962	13	1005.06	⊤	full
hex_witness_based_hein_17_6x6-13.pg	7353	1936	13	1572.7	⊥	full
hex_witness_based_hein_18_7x7-13.pg	9577	2405	13	1102.69	⊥	full
hex_witness_based_hein_20_6x6-13.pg	7551	1962	13	3123.99	⊥	full
hex_witness_based_hein_15_5x5-15.pg	7423	2136	15	2489.7	⊤	leaves
OrgSynth_mitexams_p02_l_6	83500	23384	3	1852.22	⊤	full
OrgSynth_mitexams_p02_l_7	97214	27239	3	2693.19	⊤	full
OrgSynth_mitexams_p03_l_5	106413	29730	3	2897.47	⊤	full
OrgSynth_mitexams_p07_l_5	165039	46587	3	2469.04	⊥	leaves
OrgSynth_mitexams_p16_l_6	53448	15692	3	2169.18	⊤	full
OrgSynth_mitexams_p16_l_7	62141	18265	3	3054.75	⊤	leaves
OrgSynth_mitexams_p19_l_6	106252	29346	3	3489.44	⊤	full
OrgSynth_mitexams_p20_l_7	74375	21534	3	1782.51	⊥	full
OrgSynth_mitexams_p01_l_4	65294	17864	3	1609.48	⊥	full
OrgSynth_mitexams_p05_l_3	79279	22897	3	2055.46	⊤	leaves
OrgSynth_mitexams_p05_l_4	105042	30409	3	2253.59	⊤	full
OrgSynth_mitexams_p10_l_3	44309	12864	3	870.16	⊤	full
OrgSynth_mitexams_p10_l_4	58490	17046	3	2163.5	⊤	full
OrgSynth_mitexams_p13_l_3	52653	14953	3	1310.32	⊤	full
OrgSynth_mitexams_p13_l_4	69554	19819	3	2592.6	⊤	leaves
OrgSynth_sat18_p09_l_3	52653	14953	3	1765.8	⊤	leaves
OrgSynth_sat18_p09_l_4	69554	19819	3	2328.99	⊤	leaves
OrgSynth_sat18_p11_l_4	85537	23860	3	2123.52	⊥	leaves
OrgSynth_sat18_p12_l_4	82734	23155	3	2803.72	⊥	leaves

Table 1: 48 instances that were only solved by a ParaQooba configuration. QA: Quantifier Alternations, Res: Result, Variant: ParaQooba configuration that solved the problem the fastest (preprocess full formula, preprocess leaves, original formula).

B Instances Solved faster by ParaQooba

Name	PQ [s]	VPS [s]	Speedup	Res
nreachq_query71_1344n	2.21	39.97	18.09	⊥
hex_witness_based_hein_08_5x5-11.pg	0.22	3.88	17.64	⊤
mult9.sat	2.11	34.73	16.46	⊤
add5_COMPLETE	1.78	26.98	15.16	⊤
hex_symbolic_explicit_goal_hein_10_5x5-11.pg	32.23	465.43	14.44	⊥
hex_compact_goal_witness_based_hein_10_5x5-13.pg	144.98	1853.09	12.78	⊤
hex_symbolic_explicit_goal_hein_11_5x5-09.pg	1.79	22.53	12.59	⊥
hex_symbolic_implicit_goal_hein_03_6x6-11.pg	47.52	538.03	11.32	⊥
reachqu_query60_1344n	7.57	77.4	10.22	⊥
query71_query36_1344n	11.38	105.83	9.30	⊥
hex_symbolic_explicit_goal_hein_08_5x5-09.pg	1.18	10.94	9.27	⊥
hex_symbolic_implicit_goal_hein_20_6x6-11.pg	140.49	1282.38	9.13	⊥
hex_witness_based_hein_06_4x4-11.pg	3.41	30.9	9.06	⊥
hex_compact_goal_witness_based_hein_10_5x5-11.pg	13.97	121.04	8.66	⊥
hex_symbolic_implicit_goal_hein_19_5x5-11.pg	1.69	14.29	8.46	⊤
hex_symbolic_implicit_goal_hein_16_5x5-11.pg	22.26	184.75	8.30	⊥
sortnetsort10.AE.stepl.008	13.33	107.07	8.03	⊥
add7_REDUCED	135.58	1051.44	7.76	⊤
reachqu_query64_1344n	128.4	982.54	7.65	⊥
hex_compact_goal_witness_based_hein_02_5x5-11.pg	39.04	295.57	7.57	⊥
amba4b9y.unsat	10.9	81.72	7.50	⊥
hex_symbolic_implicit_goal_hein_15_5x5-13.pg	95.67	714.78	7.47	⊥
hex_compact_goal_witness_based_hein_15_5x5-13.pg	167.18	1229.74	7.36	⊥
hex_symbolic_implicit_goal_hein_06_4x4-11.pg	1.32	9.67	7.33	⊥
hex_compact_goal_witness_based_hein_16_5x5-13.pg	372.26	2713.59	7.29	⊤

Table 2: Instances that ParaQooba (PQ) solved faster compared to a virtual portfolio solver (VPS) that also solved the same problem, ordered by the relative speedup and limited to the top 25 entries. Res: Result, Speedup: $\frac{\text{VPS}[s]}{\text{PQ}[s]}$.

References

1. Balyo, T., Lonsing, F.: HordeQBF: A modular and massively parallel QBF solver. In: Creignou, N., Berre, D.L. (eds.) Proc. of the 19th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer

Science, vol. 9710, pp. 531–538. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_33
2. Beyersdorff, O., Janota, M., Lonsing, F., Seidl, M.: Quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1177–1221. IOS Press (2021). https://doi.org/10.3233/FAIA201015
3. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) Proc. of the 23rd Int. Conf. on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 6803, pp. 101–115. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_10
4. Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate quantified boolean formulae. In: Kautz, H.A., Porter, B.W. (eds.) Proc. of the 17th Nat. Conf. on Artificial Intelligence and 12th Conf. on on Innovative Applications of Artificial Intelligence (AAAI/IAAI). pp. 285–290. AAAI Press / The MIT Press (2000), http://www.aaai.org/Library/AAAI/2000/aaai00-044.php
5. Frioux, L.L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In: Vojnar, T., Zhang, L. (eds.) Proc. of the 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 11427, pp. 135–151. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_8
6. Heisinger, M.: Distributed SAT & QBF solving: The paracooba framework. Master Thesis, JKU Linz (2021)
7. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with paracooba. In: Pulina, L., Seidl, M. (eds.) Proc. of the 23rd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 12178, pp. 114–122. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_9
8. Heisinger, M., Seidl, M., Biere, A.: Artifact for Paper ParaQooba: A Fast and Flexible Framework for Parallel and Distributed QBF Solving (Nov 2022). https://doi.org/10.5281/zenodo.7554207
9. Heisinger, M., Seidl, M., Biere, A.: QuAPI: Adding assumptions to non-assuming SAT & QBF solvers. In: Konev, B., Schon, C., Steen, A. (eds.) Proc. of the Workshop on Practical Aspects of Automated Reasoning (FLoC/IJCAR). CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022), http://ceur-ws.org/Vol-3201/paper1.pdf
10. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. J. Artif. Intell. Res. **53**, 127–168 (2015). https://doi.org/10.1613/jair.4694
11. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Proc. of the 7th Int. Conf. on Hardware and Software: Verification and Testing (HVC). Lecture Notes in Computer Science, vol. 7261, pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
12. Hoos, H.H., Peitl, T., Slivovsky, F., Szeider, S.: Portfolio-based algorithm selection for circuit QBFs. In: Hooker, J.N. (ed.) Proc. of the 24th Int. Conf. on Principles and Practice of Constraint Programming (CP). Lecture Notes in Computer Science, vol. 11008, pp. 195–209. Springer (2018). https://doi.org/10.1007/978-3-319-98334-9_13
13. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Proc. of

the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 7317, pp. 114–128. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_10
14. Jordan, C., Kaiser, L., Lonsing, F., Seidl, M.: MPIDepQBF: Towards parallel QBF solving without knowledge sharing. In: Sinz, C., Egly, U. (eds.) Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 8561, pp. 430–437. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_32
15. Kaufmann, D., Kauers, M., Biere, A., Cok, D.: Arithmetic verification problems submitted to the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, p. 49. University of Helsinki (2019)
16. Lewis, M., Schubert, T., Becker, B., Marin, P., Narizzano, M., Giunchiglia, E.: Parallel QBF solving with advanced knowledge sharing. Fundam. Informaticae **107**(2-3), 139–166 (2011). https://doi.org/10.3233/FI-2011-398
17. Lonsing, F., Egly, U.: DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) Proc. of the 26th Int. Conf. on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 10395, pp. 371–384. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_23
18. Lonsing, F., Seidl, M.: Parallel solving of quantified boolean formulas. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 101–139. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_4
19. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 133–182. IOS Press (2021). https://doi.org/10.3233/FAIA200987
20. Mayer-Eichberger, V., Saffidine, A.: Positional games and QBF: The corrective encoding. In: Pulina, L., Seidl, M. (eds.) Proc. of the 23rd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 12178, pp. 447–463. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_31
21. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artif. Intell. **274**, 224–248 (2019). https://doi.org/10.1016/j.artint.2019.04.002
22. Pulina, L., Seidl, M., Shukla, A.: QBFEval 2022. http://www.qbflib.org/qbfeval22.php (2022)
23. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD). pp. 136–143. IEEE (2015)
24. Sanders, P., Schreiber, D.: Mallob: Scalable SAT solving on demand with decentralized job scheduling. J. Open Source Softw. **7**(77), 4591 (2022). https://doi.org/10.21105/joss.04591
25. Shaik, I., Mayer-Eichberger, V., van de Pol, J., Saffidine, A.: Implicit state and goals in QBF encodings for positional games (extended version) (2023). https://doi.org/10.48550/ARXIV.2301.07345
26. Shukla, A., Biere, A., Pulina, L., Seidl, M.: A survey on applications of quantified boolean formulas. In: Proc. of the 31st IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI). pp. 78–84. IEEE (2019). https://doi.org/10.1109/ICTAI.2019.00020

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.