Chapter 1
Parallel Solving of Quantified Boolean Formulas

Florian Lonsing and Martina Seidl

Abstract Quantified Boolean formulas (QBFs) extend propositional logic by universal and existential quantifiers over the propositional variables. In the same way as the satisfiability problem of propositional logic is the archetypical problem for NP, the satisfiability problem of QBFs is the archetypical problem for PSPACE. Hence, QBFs provide an attractive framework for encoding many applications from verification, artificial intelligence, and synthesis, thus motivating the quest for efficient solving technology. Already in the very early stages of QBF solving history, attempts have been made to parallelize the solving process, either by splitting the search space or by portfolio-based approaches. In this chapter, we review and compare approaches for solving QBFs in parallel.

1.1 Introduction

Since the late 1990s, there has been impressive progress in research on solving the propositional satisfiability problem (SAT) (see Chapter ??). The boost in the performance of SAT solvers enabled routine applications of SAT to large-scale industrial problems [13, 19, 85]. In practice, nowadays SAT solvers are capable of solving formulas containing hundreds of thousands of variables

Florian Lonsing
Institute of Information Systems, TU Wien,
Favoritenstr. 9-11, 1040 Wien, Austria, e-mail: florian.lonsing@tuwien.ac.at,
Supported by the Austrian Science Fund (FWF) under grant S11409-N23.

Martina Seidl
Institute for Formal Models and Verification, JKU Linz,
Altenbergerstr. 69, 4040 Linz, Austria, e-mail: martina.seidl@jku.at,
Supported by the Austrian Science Fund (FWF) under grant S11408-N23.

and millions of clauses. This is in contrast to the computational intractability that follows from the NP-completeness of SAT.

Motivated by the success story of SAT solving, problems from complexity classes beyond NP became the focus of intensive research.[1] The *polynomial hierarchy* [64, 82] is a theoretical framework to describe the complexity of problems beyond NP. Examples of problems in the polynomial hierarchy are conformant planning [75], problems related to answer set programming [27], or the computation of minimal unsatisfiable subformulas (MUSes) [54].

A natural extension of SAT is QSAT, the satisfiability problem of *quantified Boolean formulas* (QBFs) [32, 47]. In a nutshell, QBFs are propositional formulas that additionally may contain existential ($\exists$) and universal ($\forall$) quantifiers over the propositional variables. QBFs can be used to encode any problem in the polynomial hierarchy. For example, the QBFs

$$\forall x \exists y.((x \vee \neg y) \wedge (\neg x \vee y)) \tag{1.1}$$

and

$$\exists y \forall x.((x \vee \neg y) \wedge (\neg x \vee y)) \tag{1.2}$$

encode the equivalence of the variables x and y by the propositional CNF $((x \vee \neg y) \wedge (\neg x \vee y))$ under the quantifier prefixes $\forall x \exists y$ and $\exists y \forall x$, respectively. Intuitively, the QBF 1.1 asks whether for all possible assignments of variable x there exists an assignment of variable y such that the propositional CNF evaluates to true. In contrast to that, the QBF 1.2 asks whether there exists an assignment of y such that for all assignments of x the propositional CNF evaluates to true.

Like in propositional logic, the variables in a QBF are interpreted over the Boolean domain. Obviously, the QBF 1.1 is satisfiable since the assignment of the existential variable y can be selected depending on the assignment of the universal variable x in order to satisfy the CNF. The QBF 1.2 is unsatisfiable since it differs from the QBF 1.1 in the ordering of the variables in the quantifier prefix. Due to the ordering, in the QBF 1.2 the value of y is fixed for any value of x. Hence, in general the ordering of variables in the quantifier prefix impacts the satisfiability of a QBF.

When solving a propositional formula using a SAT solver, the solver can stop as soon as it finds an assignment to the variables which satisfies the formula. When solving a QBF, however, finding one assignment which satisfies its propositional part is not enough to show the satisfiability of the QBF. The presence of universal and existential quantifiers in a QBF and the ordering of variables in the quantifier prefix give rise to tree-shaped (counter)models for witnessing (un)satisfiability. These (counter)models represent the different choices of variable assignments that have to be made depending on the quantifier types and the ordering of variables. Figure 1.1 shows a model of the QBF 1.1 and a countermodel of the QBF 1.2.

[1] BeyondNP research community website (June 2017): `http://beyondnp.org/`

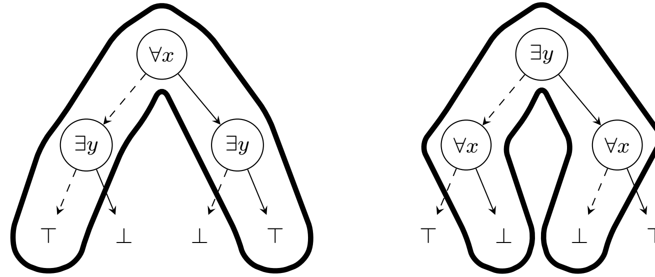

Fig. 1.1 Tree-shaped model (left) and countermodel (right) illustrating the satisfiability of the QBF 1.1 and the unsatisfiability of the QBF 1.2, respectively. (Counter)models are special subtrees of a formula's assignment tree. Dashed (solid) edges indicate that the variable in the source node is set to false (true). In the model every assignment along a path satisfies the propositional CNF $((x \vee \neg y) \wedge (\neg x \vee y))$ of the QBF, whereas in the countermodel all such assignments falsify the CNF

In practice, tree-shaped models (countermodels) are represented as functions that provide strategies to assign the existential (universal) variables with respect to universal (existential) ones if the considered formula is satisfiable (unsatisfiable). For the QBF 1.1 the function $f_y(x) := x$ yields the strategy to assign variable y to the same value as the previously assigned variable x in order to satisfy the CNF $((x \vee \neg y) \wedge (\neg x \vee y))$. Thus function $f_y(x)$ witnesses the satisfiability of the QBF 1.1. In a dual way, the function $f_x(y) := \neg y$ is a witness for the unsatisfiability of the QBF 1.2 as it represents a strategy to assign x to the opposite value of y in order to falsify the CNF.

As a consequence of extending propositional logic with quantifiers over the propositional variables to obtain the language of QBFs, the QBF satisfiability problem becomes PSPACE-complete [81]. The use of quantifiers allows for QBF encodings of problems that potentially are exponentially more succinct than the corresponding SAT encodings. Due to this property, QBFs are an attractive language for encoding and solving many practically relevant problems from domains such as, for example, formal verification [40], synthesis [14], or artificial intelligence [26, 75] (see [10] for a detailed survey). For solving such problems in practice, efficient QBF solvers are highly desirable.

Since the year 2000, there has been substantial progress in the development of efficient QBF solvers. Traditional QBF solvers can be classified into one of two dominant solving paradigms that have emerged: (1) search-based solving and (2) expansion-based solving.

Search-based QBF solvers implement a QBF-specific extension of the DPLL algorithm [18, 22] and of *conflict-driven clause learning (CDCL)* [66, 78, 79, 90]. CDCL is at the core of most modern SAT solvers. The QBF variant of CDCL, often referred to as QCDCL [35, 50, 91] implicitly searches for a tree-shaped (counter)model of the given QBF in the search space of all possible variable assignments. Thereby assignments encountered during the search that falsify the propositional part of the QBF, called *conflicts*, and assignments satisfying

it, called *solutions*, are analysed. The analysis of conflicts and solutions allows the solver to learn new *clauses* (disjunctions of literals) and *cubes* (conjunctions of literals) for pruning the search space. QCDCL solvers may apply additional QBF-specific techniques such as duality-aware reasoning [36, 37] or the analysis of variable dependencies with respect to the quantifier structure of a QBF [57].

In contrast to SAT, where CDCL is almost the single dominant solving paradigm in practice, QCDCL-based QBF solving is complemented by *expansion-based QBF solving*. Expansion-based solvers rewrite a given QBF to a satisfiability-equivalent propositional formula by successively expanding the quantifiers [5, 12]. To counter the potential exponential blow-up of the formula size that may result from expansions, *counterexample-guided abstraction refinement (CEGAR)* [20] has proven to be powerful [42, 73]. With a CEGAR-based approach to expansion, the solver operates on an abstract representation of the formula and expands quantifiers lazily. This way, only those quantifiers are expanded which promise to be useful for solving the formula. Expansion has been found to be orthogonal to QCDCL from a proof complexity perspective [11, 43].

In SAT solving, typically expansion is not applied as a standalone approach to solving since the size of formulas to be solved is often prohibitive. Instead expansion is used as a pre- and inprocessing technique in a resource-bounded way [23, 45]. Inprocessing is an approach where preprocessing is dynamically interleaved with the search process in CDCL. Bounded expansion has also been applied successfully for QBF preprocessing [17].

Given the recently published literature on QBF solving, today the main focus of QBF solver development is still on sequential systems. In QBF solving there is no general consensus on which solving paradigm is superior in practice as the performance of solvers may be highly sensitive to the considered benchmarks (for example, see the results of related QBF evaluations and competitions [41, 60, 71]). The landscape of sequential QBF solving has changed and evolved as novel solving approaches have emerged, such as nested SAT solving [15], clause selection [44, 73] or the computation of functions that represent strategies of QBFs [72].

While parallelization is natural for most QBF-solving approaches, it also introduces additional complexity in solver engineering and development. Therefore it is not surprising that solver developers first focus on the implementation of stable and efficient sequential systems before facing the challenge of parallelization. Nevertheless, since the beginning of QBF solving in the early 2000s, several approaches have been investigated to parallelize QBF solvers and thus benefit from modern clusters and multicore processors. After a period of relatively little progress, the interest in parallel QBF solving has increased, which is reflected by the QBF competition *QBFEVAL'16* [71] held in 2016.[2] There, five different parallel solvers in six configurations participated, while

[2] QBFEVAL'16 website: `http://www.qbflib.org/qbfeval16.php`

in previous competitions the parallel-solving track had to be canceled due to the lack of participants.

In this chapter, we give an overview of previous and recent approaches to parallel QBF solving. To this end, we first review the necessary preliminaries related to QBFs and recapitulate relevant sequential-solving approaches. On this basis we first present general ideas of parallelization and then introduce and compare concrete approaches implemented in parallel QBF solvers. Finally, we conclude this chapter with a selection of challenges that have to be faced in order to make parallel QBF solving ready for applications in practice.

1.2 Background

In this section, we recapitulate syntax and semantics of QBFs and summarize the terminology used in the rest of this chapter.

The language $\mathcal{L}_\mathcal{V}$ of quantified Boolean formulas over a set of propositional variables $\mathcal{V}$ and truth constants $\top$ and $\bot$ is defined as the smallest set such that

1. if $x \in (\mathcal{V} \cup \{\top, \bot\})$ then $x \in \mathcal{L}_\mathcal{V}$;
2. if $\phi \in \mathcal{L}_\mathcal{V}$ then $\neg\phi \in \mathcal{L}_\mathcal{V}$;
3. if $\phi_1, \phi_2 \in \mathcal{L}_\mathcal{V}$ then $(\phi_1 \circ \phi_2) \in \mathcal{L}_\mathcal{V}$ where $\circ \in \{\vee, \wedge, \rightarrow, \leftrightarrow, \oplus\}$;
4. if $\phi \in \mathcal{L}_\mathcal{V}$ and $x \in \mathcal{V}$, then $(\mathsf{Q}x.\phi) \in \mathcal{L}_\mathcal{V}$ where $\mathsf{Q} \in \{\forall, \exists\}$.

If convenient and unambiguous, we omit parenthesis in QBFs $\phi \in \mathcal{L}_\mathcal{V}$. For a QBF $\mathsf{Q}x.\phi$, ϕ is the *scope* of the quantifier $\mathsf{Q}x$. A variable x is *free* in a QBF ϕ, if x does not occur in the scope of a quantifier $\mathsf{Q}x$ in ϕ. A QBF is *closed* if it does not contain any free variables. In the following, we consider only closed QBFs. Furthermore, we assume that for each $x \in \mathcal{V}$, a QBF contains at most one occurrence of $\mathsf{Q}x$. For $\exists x_1, \ldots, \exists x_n$ and $\forall y_1, \ldots, \forall y_n$ we also write $\exists X$ and $\forall Y$, respectively, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$. We define $\mathsf{var}(\phi) := \{x \mid \mathsf{Q}x \text{ occurs in } \phi, \mathsf{Q} \in \{\forall, \exists\}\}$.

A *literal* is a propositional variable $x \in \mathcal{V}$ or its negation $\neg x$. By $\bar{l}$ we denote the negation of literal l. Further, $\mathsf{var}(l) := x$ if $l = x$ or $l = \neg x$. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. A clause (cube) C is *tautological* (*contradictory*) if $\{x, \neg x\} \subseteq C$. A propositional formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A propositional formula is in *disjunctive normal form* (DNF) if it is a disjunction of cubes. When convenient, we interpret a formula in CNF (DNF) as a set of clauses (cubes) and clauses (cubes) as sets of literals.

A QBF ϕ is in *prenex conjunctive normal form* (PCNF) if it has the form $\Pi.\psi$ where $\Pi := \mathsf{Q}_1 X_1, \ldots, \mathsf{Q}_n X_n$ is the prefix of ϕ and ψ is the matrix of ϕ. The matrix ψ is a propositional formula in CNF over the variables in Π. The variable sets X_i are pairwise disjoint and for $\mathsf{Q}_i \in \{\forall, \exists\}$, $\mathsf{Q}_i \neq \mathsf{Q}_{i+1}$. We define $\mathsf{var}(\Pi) := X_1 \cup \ldots \cup X_n$. The quantifier $\mathsf{quant}(\Pi, l)$ of literal l is Q_i

if $\mathsf{var}(l) \in X_i$. Given literals l and k, then $l \leq_\Pi k$ if $\mathsf{quant}(\Pi, l) = \mathsf{Q}_i$ and $\mathsf{quant}(\Pi, k) = \mathsf{Q}_j$ and $i \leq j$. For example, QBFs 1.1 and 1.2 are in PCNF.

A QBF ϕ over variables $\mathcal{V}$ is in *negation normal form* (NNF) if (1) $\phi \in \mathcal{L}_\mathcal{V}$, (2) the negation symbol occurs only directly in front of variables or truth constants, and (3) the only binary connectives are conjunction ($\wedge$) and disjunction ($\vee$). Note that the NNF structure does not impose any restrictions on the positions of quantifiers.

A *partial assignment* of the variables $\mathsf{var}(\phi)$ of a QBF ϕ is a total mapping $A\colon \mathsf{var}(\phi) \mapsto \mathbb{B} \cup \{\mathsf{U}\}$, where $\mathbb{B} := \{\mathsf{T}, \mathsf{F}\}$ is the Boolean domain and U denotes that the assignment of a variable is undefined. A *full assignment* is a total mapping $A\colon \mathsf{var}(\phi) \mapsto \mathbb{B}$. Given an assignment A of QBF ϕ, we also write A as a set of literals $A = \{l_1, \ldots, l_n\}$ such that, for all $x \in \mathsf{var}(\phi)$, $x \in A$ if $A(x) = \mathsf{T}$, $\neg x \in A$ if $A(x) = \mathsf{F}$, and both $x \notin A$ and $\neg x \notin A$ if $A(x) = \mathsf{U}$. Then for any $l_i, l_j \in A$ with $i \neq j$, $\mathsf{var}(l_i) \neq \mathsf{var}(l_j)$.

For a QBF ϕ and an assignment A, $\phi[A]$ denotes the QBF ϕ *under* A which is obtained from ϕ as follows. For all $l \in A$ with $\mathsf{var}(l) = x$, the quantifier $\mathsf{Q}x$ is removed, any occurrence of x is replaced by $\top$ if $x \in A$ and by $\bot$ if $\neg x \in A$, followed by the usual simplifications of Boolean logic. For example, if $\phi := \Pi.\psi$ is in PCNF, then for all $l \in A$ any clause C with literal $l \in C$ is deleted, any occurrence of literal $\bar{l}$ is removed, and the variable $\mathsf{var}(l)$ of l and its quantifier $\mathsf{quant}(\Pi, l)$ are removed from the prefix. If $\phi[A]$ simplifies to $\top$ (written as $\phi[A] = \top$) then A is called a *satisfying assignment*. If $\phi[A]$ simplifies to $\bot$ (written as $\phi[A] = \bot$) then A is called a *falsifying assignment*.

An *assignment tree* of a QBF ϕ is a complete binary tree of depth $|\mathsf{var}(\phi)|+1$ where the internal nodes of each level are associated with a variable of ϕ. The levels reflect the order of the quantifiers in the formula. The outgoing edges of an internal node labeled by variable x are associated with $\neg x$ and x, indicating that x is set to false and to true, respectively. A path from the root of the tree to a leaf represents a particular variable assignment. The leaves are labeled by the truth value of ϕ under the assignment of the respective path. Figure 1.1 shows examples of two assignment trees. The highlighted subtrees of the assignment trees represent a model and a countermodel, respectively.

The *semantics* of QBFs is defined recursively based on the syntactic structure as follows. The QBF $\phi := \top$ is satisfiable and the QBF $\phi := \bot$ is unsatisfiable. A QBF $\forall x.\phi$ is satisfiable iff $\phi[x]$ is satisfiable and $\phi[\neg x]$ is satisfiable. A QBF $\exists x.\phi$ is satisfiable iff $\phi[x]$ is satisfiable or $\phi[\neg x]$ is satisfiable. The Boolean connectives are interpreted according to standard semantics. Two QBFs ϕ and ϕ' are *satisfiability-equivalent* iff ϕ is satisfiable whenever ϕ' is satisfiable.

Example 1. The QBF 1.1 $\phi := \forall x \exists y.((x \vee \neg y) \wedge (\neg x \vee y))$ is satisfiable since both $\phi[\{\neg x\}] = \exists y.(\neg y)$ and $\phi[\{x\}] = \exists y.(y)$ are satisfiable. In contrast to that, the QBF 1.2 $\phi := \exists y \forall x.((x \vee \neg y) \wedge (\neg x \vee y))$ is unsatisfiable since neither $\phi[\{\neg y\}] = \forall x.(\neg x)$ nor $\phi[\{y\}] = \forall x.(x)$ is satisfiable.

In the following, we define the *Q-resolution calculus*, the formal framework of QBF solvers based on QCDCL. The calculus consists of rules that allow us to derive clauses and cubes from a given PCNF ϕ. The implementation of clause (cube) learning in QCDCL relies on the Q-resolution calculus.

Definition 1 (Q-Resolution Calculus [35, 48, 50, 91]). Let $\phi = \Pi.\psi$ be a formula in PCNF. The rules of the *Q-resolution calculus* are as follows:

$$\frac{C \cup \{l\}}{C} \quad \begin{array}{l} \text{if for all } x \in \mathsf{var}(\Pi)\colon \{x, \bar{x}\} \not\subseteq (C \cup \{l\}) \text{ and either} \\ \text{(1) } C \text{ is a clause, } \mathsf{quant}(\Pi, l) = \forall, \\ \quad l' <_{\Pi} l \text{ for all } l' \in C \text{ with } \mathsf{quant}(\Pi, l') = \exists \text{ or} \\ \text{(2) } C \text{ is a cube, } \mathsf{quant}(\Pi, l) = \exists, \\ \quad l' <_{\Pi} l \text{ for all } l' \in C \text{ with } \mathsf{quant}(\Pi, l') = \forall \end{array} \qquad (red)$$

$$\frac{C_1 \cup \{p\} \qquad C_2 \cup \{\bar{p}\}}{C_1 \cup C_2} \quad \begin{array}{l} \text{if for all } x \in \mathsf{var}(\Pi)\colon \{x, \bar{x}\} \not\subseteq (C_1 \cup C_2), \\ \bar{p} \notin C_1,\ p \notin C_2, \text{ and either} \\ \text{(1) } C_1,\ C_2 \text{ are clauses, } \mathsf{quant}(\Pi, p) = \exists \text{ or} \\ \text{(2) } C_1,\ C_2 \text{ are cubes, } \mathsf{quant}(\Pi, p) = \forall \end{array} \qquad (res)$$

$$\frac{}{C} \quad \begin{array}{l} A \text{ is an assignment, } \phi[A] = \top, \\ \text{and } C = (\bigwedge_{l \in A} l) \text{ is a cube} \end{array} \qquad (cu\text{-}init)$$

$$\frac{}{C} \quad \text{if for all } x \in \mathsf{var}(\Pi)\colon \{x, \bar{x}\} \not\subseteq C \text{ and } C \in \psi \text{ is a clause} \qquad (cl\text{-}init)$$

A QBF ϕ in PCNF is unsatisfiable (satisfiable) [35, 48, 50, 91] iff the empty clause (empty cube) $\emptyset$ is derivable from ϕ by applying the rules given in Def. 1. A derivation of the empty clause (cube) $\emptyset$ from ϕ starting with applications of the axiom rules *cl-init* (*cu-init*) is a *Q-resolution proof* of the unsatisfiability (satisfiability) of ϕ.

In the case of unsatisfiability, non-tautological clauses occurring in ϕ are selected by applications of axiom rule *cl-init*. In the case of satisfiability, cubes obtained from satisfying assignments are derived by applications of axiom rule *cu-init*.

The variants of rule *res* to resolve clauses or cubes, respectively, are similar to the resolution rule in propositional logic. In this chapter, we assume that the *pivot variable* p is existential (universal) when resolving clauses (cubes) by rule *res*. Furthermore, clauses (cubes) derived by *res* must not be tautological (contradictory). These restrictions define the most common variant of Q-resolution [48]. However, it has been shown that the restriction may be lifted, resulting in more powerful variants of Q-resolution [7, 29].

The main distinguishing feature between propositional resolution and Q-resolution is rule *red*, the *reduction* operation. *Universal (Existential) reduc-*

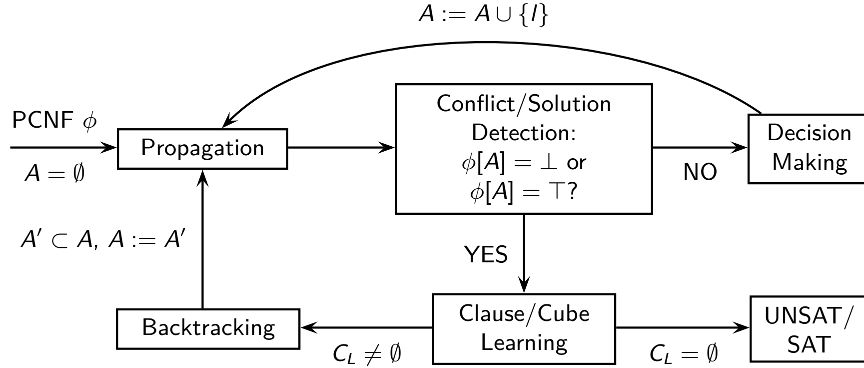

Fig. 1.2 Flowchart of QCDCL (adapted from [59]). Stages *propagation* and *conflict/solution detection* are part of function `qbcp` in Algorithm 1.1, and stages *clause/cube learning* and *backtracking* are part of function `analyze`

tion eliminates trailing universal (existential) literals from a non-tautological clause (non-contradictory cube) C with respect to the quantifier ordering. We write $\mathsf{UR}(C) = C'$ ($\mathsf{ER}(C) = C'$) to denote the clause (cube) C' resulting from clause (cube) C by universal (existential) reduction. For a PCNF $\phi = \Pi.\psi$, $\mathsf{UR}(\phi) = \Pi.(\bigwedge_{C \in \psi} \mathsf{UR}(C))$ is the PCNF resulting from universal reduction of every clause $C \in \psi$.

1.3 Sequential Search-Based QBF Solving

Most parallel QBF solvers are based on the search-based QBF-solving paradigm. Therefore, we briefly recapitulate the core concepts and ideas behind search-based solvers.

Search-based QBF solving [18] lifts the DPLL algorithm [22] to QBF. Conflict-driven clause learning (CDCL) in SAT solving [66, 78, 79, 80, 90] extends DPLL by clause learning. Clauses are learned from conflicts to prune the search space during the search for a satisfying assignment. The QBF-specific variant of CDCL is usually called QCDCL [35, 50, 91]. In contrast to CDCL-based SAT solvers, QCDCL-based QBF solvers not only learn clauses from conflicts, but also cubes from solutions. Clauses and cubes are learned using the rules of the Q-resolution calculus. The pseudocode in Algorithm 1.1 and the flowchart in Figure 1.2 provide a high-level description of QCDCL.

From a high-level point of view, the basic building blocks of QCDCL such as propagation, decision making, learning, and backtracking are similar to CDCL. Given an input formula $\phi = \Pi.\psi$ in PCNF, the assignment tree of ϕ is traversed in a depth-first manner. QCDCL terminates if the empty clause

(cube) is derived in clause (cube) learning, which shows the unsatisfiability (satisfiability) of ϕ.

Learned clauses and cubes are stored in separate sets ϕ_{CL} and ϕ_{CU}, respectively. In practice, the set of learned clauses ϕ_{CL} is added conjunctively to $\phi = \Pi.\psi$ to obtain the satisfiability-equivalent formula $\Pi.(\psi \wedge (\bigwedge_{C \in \phi_{CL}} C))$. In a similar way, the set of learned cubes is added disjunctively to $\phi = \Pi.\psi$ to obtain the satisfiability-equivalent formula $\Pi.(\psi \wedge (\bigvee_{C \in \phi_{CU}} C))$. Adding learned clauses (cubes) to ϕ preserves the satisfiability (unsatisfiability) of ϕ due to the soundness of the Q-resolution calculus.

Given the current assignment A (which is initially empty), *unit* and *pure literals* are detected and assigned during QBF-specific Boolean constraint propagation (called QBCP, cf. function `qbcp` in Algorithm 1.1) [18, 31]. To this end, the PCNF $\phi[A]$ is considered, i.e., ϕ interpreted under A. Some literal l is unit in $\phi[A]$ if $\phi[A]$ contains a clause (l). Some literal l is pure in $\phi[A]$ if $\bar{l}$ does not occur in $\phi[A]$. Unit and pure literal detection is also applied to the learned clauses and cubes in sets ϕ_{CL} and ϕ_{CU}, respectively. While unit clause detection is similar to CDCL, in QBCP additionally universal reduction by rule *red* of the Q-resolution calculus is applied to the clauses in $\phi[A]$.

After the techniques in QBCP have been applied until saturation, in *conflict/solution detection* (part of function `qbcp`) it is checked whether $\phi[A] = \bot$ or $\phi[A] = \top$.

If neither $\phi[A] = \bot$ nor $\phi[A] = \top$ (line 5 in Algorithm 1.1) then A is extended by tentatively assigning some variable in *decision making* (function `assign_dec_var`). A SAT solver may assign any unassigned variable of the formula. However, this would not be sound in QCDCL. Only variables from

Algorithm 1.1: Pseudocode of QCDCL

```
   Data: PCNF φ
   Result: True (false) if φ is satisfiable (unsatisfiable)
1  Result R = UNDEF;
2  Assignment A = ∅;
3  while true do
       /* Simplify under A, propagation.                    */
4      (R, A) = qbcp(φ, A);
5      if R == UNDET then
           /* Decision making.                              */
6          A = assign_dec_var(φ, A);
7      else
           /* Backtracking: R == UNSAT/SAT                  */
8          A' = analyze(R, A) ;
9          if A' == INVALID then
10             return R;
11         else
12             A = backtrack(A');
```

the outermost, i.e., leftmost quantifier block of $\phi[A]$ may be assigned as decisions. As in SAT solving, it does not affect soundness whether a variable is first set to true or to false. After a variable has been assigned in decision making, propagation continues (function `qbcp`).

If $\phi[A] = \bot$ (line 7 in Algorithm 1.1) then ϕ (or ϕ_{CL}, respectively) contains a clause C for which $\mathsf{UR}(C[A]) = \emptyset$. This situation is called a *conflict*. Conflicts trigger clause learning, where a learned clause C_L is derived using the rules of the Q-resolution calculus. Thereby, C is successively resolved with *antecedent clauses* of unit literals identified during QBCP. The antecedent clause of a unit literal l is the clause in ϕ containing l that became unit in $\phi[A]$ during QBCP.

If $\phi[A] = \top$ (line 7 in Algorithm 1.1), then $\phi[A] = \emptyset$, i.e., ϕ reduces to the empty matrix under A, or ϕ_{CU} contains a cube C for which $\mathsf{ER}(C[A]) = \emptyset$. This situation is called a *solution*. A solution corresponds to a single path in the assignment tree of ϕ where the leaf is labeled with $\top$ (cf. Fig. 1.1). A SAT solver would terminate after a solution has been found. However, due to universally quantified variables, a QCDCL QBF solver in general must proceed and find further solutions. Solutions trigger cube learning, where a learned cube C_L is derived in a similar way to a learned clause. Cubes to be resolved on by rule *res* have to be derived by rule *cu-init* first.

Clause (cube) learning is part of function `analyze` in Algorithm 1.1. If the empty clause (cube) C_L is derived in clause (cube) learning ($C_L = \emptyset$ in Figure 1.2), then QCDCL terminates and reports the unsatisfiability (satisfiability) of the input PCNF ψ (line 10 in Algorithm 1.1).

Otherwise ($C_L \neq \emptyset$ in Figure 1.2), during backtracking the current assignment A is analyzed (line 8, function `analyze`) in order to retract a subassignment $A' \subseteq A$ of A (line 12). The subassignment A' is selected so that the learned clause (cube) becomes unit under the new assignment A that results from backtracking. Clauses (cubes) C_L having this property are called *asserting*. In QCDCL, typically only asserting clauses and cubes are learned. The run of QCDCL proceeds with the new assignment A resulting from backtracking.

Example 2 (Based on an example from [55]). Consider the PCNF $\phi = \Pi.\psi$ with prefix $\Pi = \exists z,z'\forall u\exists y$ and CNF

$$\psi = (u \lor \bar{y}) \land (\bar{u} \lor y) \land (z \lor u \lor \bar{y}) \land (z' \lor \bar{u} \lor y) \land (\bar{z} \lor \bar{u} \lor \bar{y}) \land (\bar{z}' \lor u \lor y)$$

Initially the current assignment A and the sets of learned clauses and cubes are empty. Propagation does not have any effect since ϕ does not contain unit literals (to keep the example simple, we do not carry out pure literal detection). Suppose that both z and z' are assigned *true* in decision making, i.e., $A := \{z, z'\}$, resulting in the PCNF $\phi[A] = \forall u\exists y.(u\lor\bar{y})\land(\bar{u}\lor y)\land(\bar{u}\lor\bar{y})\land(u\lor y)$. Again, $\phi[A]$ does not contain unit literals to be propagated. Hence, let A be extended by assignment $\{u\}$ in decision making, i.e., $A := \{z, z', u\}$, resulting in $\phi[A] = \exists y.(y) \land (\bar{y})$. Suppose that variable y is assigned *true* by unit literal

detection applied to $\phi[A]$, where $(\bar{u} \vee y) \in \phi$ is the antecedent clause of the derived assignment $\{y\}$. Clause $C_1 = (\bar{z} \vee \bar{u} \vee \bar{y}) \in \phi$ is falsified under $A := \{z, z', u, y\}$, i.e., $\mathsf{UR}(C_1[A]) = \emptyset$. In clause learning, C_1 is resolved with the antecedent clause $(\bar{u} \vee y)$ by pivot variable y, resulting in the asserting learned clause $C_{L,1} = (\bar{z})$ after universal reduction.

Based on the result of `analyze`, the whole current assignment $A = \{z, z', u, y\}$ is retracted to the empty assignment $A = \emptyset$. Note that, in particular, all assignments of variables due to decision making are retracted, which corresponds to non-chronological backtracking. Since the learned clause $C_{L,1}$ is unit, i.e., a clause of size one, under the empty assignment A, propagation updates A to $A := \{\bar{z}\}$. Next, suppose that z' and u are assigned as decisions to obtain $A := \{\bar{z}, \bar{z}', \bar{u}\}$. Finally we get $A := \{\bar{z}, \bar{z}', \bar{u}, \bar{y}\}$ by unit literal detection. Every clause in ϕ is satisfied under A. In cube learning, the new cube $C_2 = (\bar{z} \wedge \bar{z}' \wedge \bar{u} \wedge \bar{y})$ is derived using rule *cu-init* of the Q-resolution calculus. From C_2, the asserting learned cube $C_{L,2} = (\bar{z} \wedge \bar{z}' \wedge \bar{u}) = \mathsf{ER}(C_2)$ is derived by existential reduction (rule *red*).

After retracting $\{\bar{u}, \bar{y}\}$ from A to obtain $A := \{\bar{z}, \bar{z}'\}$ due to the result of `analyze`, $C_{L,2}$ becomes unit and hence A is extended to $A := \{\bar{z}, \bar{z}', u\}$, thus flipping the assignment of u. Cube $C_{L,2}$ is the antecedent cube of assignment $\{u\}$. Finally $A := \{\bar{z}, \bar{z}', u, y\}$ by unit clause detection. Every clause in ϕ is satisfied under A. Cube $C_3 = (\bar{z} \wedge \bar{z}' \wedge u \wedge y)$ is derived by rule *cu-init* as before and further $C_4 = (\bar{z} \wedge \bar{z}' \wedge u) = \mathsf{ER}(C_3)$ by existential reduction of C_3. Q-resolution of the antecedent cube $C_{L,2}$ of assignment $\{u\}$ and C_4 using pivot variable u produces $C_5 = (\bar{z} \wedge \bar{z}')$. Finally, existential reduction of C_5 results in the empty cube, proving that ϕ is satisfiable.

1.4 Parallel QBF Solving at a Glance

In this section, we present approaches to parallel QBF solving that have been implemented in 11 different solvers summarized in Table 1.1. Before we discuss the individual solvers in detail in the next section, we first outline the basic ideas behind the approaches. Parallel QBF solving can be classified into *portfolio* approaches and approaches based on *search space splitting*.

A conceptually simple and straightforward way to solve a QBF in parallel is the use of a *portfolio* approach. Thereby, given a set of sequential solvers having different solving characteristics or one sequential solver in different configurations, the input formula is solved by running the solver instances in parallel on separate computing nodes. The nodes may be logically separated, like in threaded solvers, or physically separated like in distributed solvers.

Due to the orthogonality of QCDCL and expansion-based QBF solving that has been witnessed both in proof complexity [11, 43] and in experimental studies [60, 63], portfolio approaches appear to be a promising direction for the implementation of parallel QBF solvers. Since QBF solving by QCDCL

and expansion has different characteristics depending on the input formula, a parallel QBF portfolio solver which combines these two solving paradigms can exploit the benefits of both approaches. However, in contrast to parallel SAT solving, where portfolio solvers are well studied and established (see Chapter **??**), few parallel portfolio QBF solvers have been presented.

We are aware of the following three parallel QBF solvers based on the portfolio approach. The solver HordeQBF [8] applies the HordeSAT framework [9] (cf. Chapter **??**) to QBF, allowing us to run different configurations of one QCDCL solver in a massively parallel manner. The solvers hiqqerfork and par-pd-depqbf are implemented as Linux shell scripts which run instances of sequential solvers in parallel processes. While hiqqerfork uses different configurations of one solver in the parallel processes, par-pd-depqbf uses two identical solver instances to solve different input formulas. To this end, par-pd-depqbf takes structured non-PCNF formulas ϕ in the QCIR format[3] as input and transforms both ϕ and its negation $\neg\phi$ into PCNF [30]. Then one process in par-pd-depqbf runs a solver instance to solve the primal PCNF encoding of ϕ, and a second process runs an identical solver instance to solve the dual PCNF encoding of $\neg\phi$.

The most widely used approach to parallel QBF solving in terms of implemented solvers, however, is based on *search space splitting* by analyzing the formula structure as follows. Consider Algorithm 1.2 which shows a very basic recursive algorithm to evaluate a QBF of arbitrary syntactic structure. In fact, this algorithm is a direct translation of the QBF semantics given in Section 1.2 into pseudocode. The evaluation of a QBF is broken down into subproblems. The base cases of the evaluation are QBFs consisting of only a truth constant. Compound formulas containing operators such as negation, binary connectives, or quantifiers are evaluated depending on the respective semantics of the operators. That is, the result of evaluating a QBF depends on the results of evaluating its subformulas.

Algorithm 1.2 already illustrates the potential of parallel QBF solving. For example, if we want to solve the QBF $\forall x.\psi$, then we can solve $\psi[x]$ and $\psi[\bar{x}]$ in parallel processes and then combine the results according to the semantics of the universal quantifier. If either $\psi[x]$ or $\psi[\bar{x}]$ is found unsatisfiable in one process, then the other process can be stopped since the given QBF $\forall x.\psi$ has been proved unsatisfiable already. The situation is similar when solving a non-PCNF formula like $\psi_1 \vee \psi_2$. The subformulas ψ_1 and ψ_2 can be solved independently by two different processes—as soon as one of the subformulas is found to be satisfiable, the process evaluating the other subproblem can be stopped due to the semantics of the $\vee$ operator.

Based on the above observations related to Algorithm 1.2, an obvious way to parallelize QBF solving is to split the problem of evaluating the original formula into several subproblems, which are then distributed to the different

[3] QCIR format: `http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf`

Algorithm 1.2: Splitting Algorithm for QBF Evaluation

```
   Data: QBF φ
   Result: True (false) if φ is satisfiable (unsatisfiable)
 1 begin
 2   switch φ do
 3     case ⊤
 4       return SAT;
 5     case ⊥
 6       return UNSAT;
 7     case ¬ψ
 8       return NOT split(ψ);
 9     case ψ1 ∨ ψ2
10       return split(ψ1) OR split(ψ2);
11     case ψ1 ∧ ψ2
12       return split(ψ1) AND split(ψ2);
13     case ∃x.ψ
14       return split(ψ[x/⊤]) OR split(ψ[x/⊥]);
15     case ∀x.ψ
16       return split(ψ[x/⊤]) AND split(ψ[x/⊥]);
```

client solvers. Either a sequential solver is called for each subproblem or the subproblem is split further.

Reconsider QBF $\phi = \exists z, z' \forall u \exists y.\psi$ from Example 2. The assignment tree of ϕ is shown in Fig. 1.3. Two processes could solve the subproblems $\phi[z]$ and $\phi[\bar{z}]$ independently and in parallel. Example 2 presented a sequential solver run in which the subproblem $\phi[z]$ was considered first, i.e., the variable z was first set to true in decision making. Only after undoing this decision in backtracking, the solver entered that part of the assignment tree that contains the model of ϕ (i.e., the left subtree in Fig. 1.3). If variable z were first set to the opposite value, i.e., false, then the extra work spent on evaluating the subproblem $\phi[z]$ would have been avoided altogether. It would not be necessary to wait for the solver to enter the part of the search space given by subproblem $\phi[\bar{z}]$, which contains the model. Moreover, if the two subproblems $\phi[\bar{z}]$ and $\phi[z]$ are solved in parallel, then the search can be stopped as soon as one subproblem witnesses the satisfiability of ϕ. If a subproblem, e.g., $\phi[\bar{z}]$ turns out to be too hard for a process to solve within certain resource limits, then it can be split again into further subproblems $\phi[\bar{z}, z']$ and $\phi[\bar{z}, \bar{z}']$, provided that the necessary computing resources are available. Again these subproblems can be solved independently of each other, and only the results of their evaluations need to be merged according to the semantics of the existential quantifier in $\exists z'$. Subproblems related to universal quantification are handled analogously.

As illustrated by Example 2, QCDCL solvers learn clauses and cubes from conflicts and solutions encountered during the search. When solving a QBF

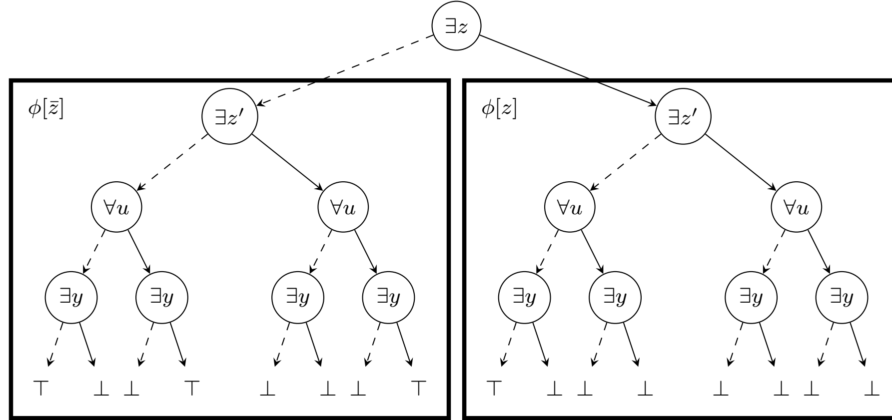

Fig. 1.3 Assignment tree of QBF ϕ from Example 2 and two subtrees of $\phi[\bar{z}]$ and $\phi[z]$

in parallel, these derived clauses and cubes potentially are helpful to other processes, even if a process had only a minor contribution to identifying a model or a countermodel for the given QBF. Therefore, sharing knowledge in terms of learned clauses and cubes with other processes is crucial in parallel QBF solving.

Research on parallel QBF solving has been focused on (1) the generation of subproblems, which are delegated to processes running on different computing nodes, and (2) knowledge sharing, i.e., the distribution of information derived by one process which is potentially useful for the others. Subproblem generation and knowledge sharing for parallel QBF solving are strongly inspired by the respective approaches to parallel SAT solving (see Chapter **??**). However, the SAT approaches cannot be ported to QBF in a straightforward way.

Generating subproblems (and hence also assembling the results returned by different processes) is complicated by the quantifier types of variables and by the order of the variables with respect to the quantifier structure of a QBF.

A variant of the *guiding path method* [88] as introduced for SAT solving has been found effective at generating subproblems in parallel QBF solving. With this method, a sequential solver instance in a separate computing node is provided with a set of *assumptions*. Assumptions are predefined variable assignments that the solver has to take into account in the solving process. This way, the subproblem that the solver has to solve is defined. Assumptions can also be understood as a special kind of decision variables the solver has to treat in a certain way. For example, the subproblems $\phi[\bar{z}]$ and $\phi[z]$ in Fig. 1.3 are defined by the sets $\{\bar{z}\}$ and $\{z\}$ of assumptions, respectively.

Typically, a master process generates sets of assumptions and distributes them to the solver instances running on the computing nodes. Based on the result of solving the subproblem, the solver may request further subproblems from the master. The master combines the results of the subproblems depend-

ing on the quantifier types of the variables assigned in the set of assumptions (cf. the splitting algorithm in Algorithm 1.2). Due to the quantifier types and the ordering of variables in the quantifier prefix, the generation of subproblems and combination of results by the master process is more complicated than in the context of SAT solving. In this respect, care has to be taken to guarantee soundness and completeness of a parallel QBF solver. For instance, in Example 2 it would be unsound to generate subproblems by assumption $\{y\}$ only since y is not at the left end of the quantifier prefix of ϕ.

In contrast to SAT solvers, QCDCL solvers operating on a PCNF ϕ not only learn clauses from conflicts but also cubes from solutions. As illustrated by Example 2, initially the set of learned cubes is empty. Hence cubes have to be derived first by rule *cu-init* of the Q-resolution calculus based on satisfying assignments. Since a satisfying assignment of ϕ must satisfy every clause in ϕ, cubes derived by rule *cu-init* tend to be large and often contain a large number of the variables in ϕ. Therefore, sharing large cubes with other solver instances in a parallel setting is challenging not only because of their size but also since large cubes tend to have only a limited pruning effect on the search space.

QBF solvers that implement parallelization by the guiding path method are MPIDepQBF, PAQuBE, PQSolve, and QMiraXT. They are distinguished by whether learning is supported or not, whether subproblems are generated by the master or by the client, and the way the result returned by the client solvers is represented. To summarize, the master process in a parallel QBF solver based on search space splitting carries out the following tasks (if supported by the respective concrete approach):

- administrate the currently distributed subproblems[4];
- maintain information about decision variables used to generate assumptions;
- request new subproblems from busy clients if there are idle clients;
- activate idle clients when new subproblems are available;
- manage information sharing among the clients;
- stop the clients if the given QBF has been solved.

In principle, the clients in a parallel solver based on search space splitting are responsible for the following tasks (if supported by the concrete approach):

- receive a subproblem (ideally in terms of assumptions);
- solve a subproblem and return the respective result to the master;
- share information with other clients;
- learn information from other clients;
- optionally generate subproblems, which are passed to the master (or to other clients);
- terminate if requested by the master.

[4] In PQSolve a client may distribute subproblems to other clients and hence becomes the master with respect to the particular subproblem.

The solvers pcaqe, PQUABS, and PQSAT also use syntactic properties of a formula to split the search space, but in a conceptually different manner to the guiding path method. These solvers are based on expansion. PQSAT extracts subformulas as subproblems, which may contain free variables. The clients processing the subproblems either eliminate the remaining quantifiers such that a propositional formula over these free variables is returned to the master, or they further split the subproblem. The solvers pcaqe and PQUABS extract a propositional formula for each quantifier block that is then used for evaluating the given QBF. Differently from the other solvers, which operate on formulas in PCNF, PQUABS operates on formulas in non-prenex form.

The rough classification of parallel QBF-solving paradigms presented above already illustrates the different approaches to leverage the power of modern computing systems. In the following section, we give a detailed description of individual parallel QBF solvers.

1.5 Parallel QBF-Solving Approaches

Table 1.1 summarizes and compares the parallel QBF-solving approaches that have been presented in the literature. The only approach not implemented is the one by Aspvall et al. [3], which is restricted to PCNFs with a maximum clause size of two and so far has been of theoretical interest only. For the other approaches implementations either are publicly available or at least experimental results have been published. Most parallel solvers are based on a sequential QBF solver such as DepQBF, QuBE, caqe, quabs, QSolve, and QSAT. Usually the sequential solvers are tightly integrated into the implementations of the clients. As the only exception, HordeQBF is based on a generic framework that allows integration of any QCDCL-based QBF solver supporting incremental solving and learning. QMiraXT implements its own QBF solver in order to be used with the MiraXT framework that was developed for parallel SAT solving. The majority of the parallel QBF solvers are based on QCDCL; only pcaqe, PQSAT, and PQUABS apply expansion-based techniques. Out of the QCDCL-based solvers, three support clause and cube sharing. All solvers apply certain simplification techniques either before the solving starts, i.e., as a preprocessing step, or dynamically as *inprocessing* [45] during solving, like DepQBF as used in HordeQBF. In the following, we discuss the individual solving approaches in detail.

Approach by Aspvall et al.

One of the first parallel approaches to QBF solving was presented by Aspvall et al. in 1996 [3]. The considered QBFs ϕ are in PCNF with a quantifier prefix having arbitrarily many quantifier alternations but with the restriction

Table 1.1 Comparison of parallel QBF solvers

parallel QBF solver	*base solver*	*QCDCL-based*	*portfolio*	*PCNF input format*	*information sharing*	*pre-/improcessing*	*process management*	*QBFEVAL'16*	*publicly available*	*most recent paper*
Aspvall et al.	n.i.									1996 [3]
pcaqe	caqe[1]	×	×	✓	×	✓	Pthreads	✓	✓	–
hiqqerfork	DepQBF	✓	✓	✓	×	✓	`fork`	✓	–	–
HordeQBF	DepQBF[2]	✓	✓	✓	✓	✓	MPI	✓	✓	2016 [8]
MPIDepQBF	DepQBF	✓	×	✓	×	✓	MPI	✓	✓	2014 [46]
par-pd-depqbf	DepQBF	✓	✓	✓	×	✓	`fork`	✓	–	–
PAQuBE	QuBE	✓	×	✓	✓	✓	MPI	×	×	2011 [51]
PQSolve	QSolve	∼[4]	×	✓	×	✓	MPI	×	×	2000 [28]
PQSAT	QSAT	×	×	×	×	✓	MPI	×	×	2010 [67]
PQUABS	quabs	×	×	×	×	✓	Pthreads	×	✓	2016 [83]
QMiraXT	MiraXT[3]	✓	×	✓	✓	✓	Pthreads	×	✓	2009 [52]

✓ yes/supported × no/not supported – unpublished n.i. not implemented
[1] Picosat is the default SAT solver; also Minisat is supported
[2] any QCDCL solver could be used
[3] parallel SAT-solving framework
[4] DPLL-based

that clauses contain at most two literals. Formulas of this kind are also called *Q2CNF formulas.* In consequence, the satisfiability problem of Q2CNFs is not PSPACE-complete any more. Instead, the satisfiability of a Q2CNF ϕ can be decided by a sequential algorithm [4] in time $O(n+m)$, where n is the number of variables and m is the number of clauses in ϕ.

In principle, the approach by Aspvall et al. builds on the linear time sequential algorithm to solve Q2CNFs [4]. Let $G(\phi) := (V, E)$ be the directed *implication graph* of a Q2CNF formula $\phi = Q_1 x_1 \dots Q_n x_n.\psi$ where the set $V = \{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ of vertices is given by all possible literals $x_i \in \phi$, and for any clause $(l \vee k) \in \phi$ it holds that $(\bar{l}, k)$ and $(\bar{k}, l)$ are edges in E. A vertex of $G(\phi)$ is called existential (universal) if its associated variable is existentially (universally) quantified. Given a Q2CNF ϕ and the related graph $G(\phi)$, ϕ is satisfiable iff none of the following three conditions holds [4]:

1. Existential vertices l and $\bar{l}$ are in the same strongly connected component of $G(\phi)$.
2. A strongly connected component of $G(\phi)$ contains universal vertex l and existential vertex k with $k < l$.
3. There is a path between two universal vertices.

To test the satisfiability of a Q2CNF ϕ, first the transitive closure of $G(\phi)$ is represented as an adjacency matrix. Then the above conditions are checked

in constant parallel time by assigning one processor to each pair of variables. Furthermore, Aspvall et al. present an algorithm to find models of satisfiable Q2CNFs: because of the restricted formula structure it is sufficient that the values of the existential variables are mapped either to truth constants or to one universal literal.

In the original publication [3] no implementation of the algorithm was reported, and we are not aware of any implementation published elsewhere. In practical QBF applications, encodings of problems typically have clauses of size bigger than two. Therefore it is unlikely that this approach will ever be implemented in a dedicated parallel Q2CNF solver. However, in the same way as the sequential version [4] of this algorithm is used to identify equivalent literals (e.g., in the preprocessor bloqqer [38]), also its parallel variant could be used for speeding up preprocessing.

Unpublished QBFEVAL'16 participants: pcaqe, hiqqerfork, par-pd-depqbf

Three parallel solvers not formally published in the literature participated in the parallel track of QBFEVAL'16. These solvers are par-pd-depqbf, hiqqerfork, and pcaqe, which solved the largest number of formulas in the parallel track, i.e., 606, 598, and 585 formulas out of 825, respectively [71]. We briefly review these solvers in the following.

Both hiqqerfork and par-pd-depqbf may be considered to be portfolio-based solvers. The solver hiqqerfork is a portfolio solver in the classical sense, running different configurations of the sequential solver hiqqer. A short description of hiqqer can be found in [41]. The solver hiqqer uses modifications of the publicly available preprocessors bloqqer and qxbf before invoking the solver DepQBF.

The solver par-pd-depqbf is based on the insight that often it is not clear whether the *primal* or the *dual* encoding of a problem is preferable for a particular solver [30]. The primal encoding represents the original problem, whereas the dual encoding represents its negation. The solver par-pd-depqbf runs exactly two identical instances of a sequential QBF solver in parallel. Given a structured non-PCNF formula ϕ in the QCIR format as input, one solver instance processes the primal encoding of ϕ as a PCNF, and the other solver instance processes the dual encoding of $\neg\phi$ as a PCNF. If either the primal or the dual version is solved, the whole solving process is stopped and the respective result is returned. The sequential back-end solver of par-pd-depqbf is DepQBF in combination with the preprocessor bloqqer. However, basically any QBF solver or preprocessor can be applied in par-pd-depqbf.

The solver pcaqe is a parallel version of the sequential solver caqe [73] which is based on a similar abstraction-based technique to that used in the

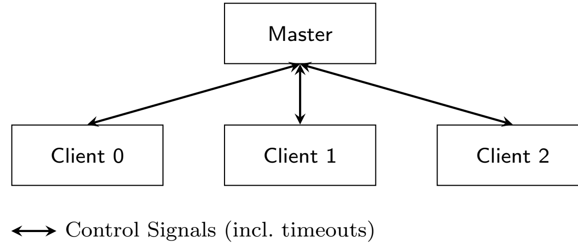

Fig. 1.4 Master-Client architecture of MPIDepQBF

solver PQUABS (see below). The solver pcaqe is part of the source code of caqe[5] and can be run with either Minisat or Picosat as back-end solver.

MPIDepQBF

The solver MPIDepQBF [46] relies on the sequential QCDCL-based solver DepQBF to solve any input formula ϕ in PCNF. To this end, ϕ is split into subproblems to be evaluated by client processes operating in parallel. The clients are independent of each other and do not exchange any information. However, information learned locally by a client is reused in different runs of that client. Keeping the information from run to run is realized by assumption-based reasoning. Assumptions are temporary (and partial) assignments of variables that define the formula to be solved by a client following the guiding path method.

In MPIDepQBF one dedicated master process coordinates an arbitrary number of clients via MPI (see Fig. 1.4). The sequential solver DepQBF applied by the clients provides an API similar to the APIs of most incremental SAT solvers [25, 68]. The API allows the solver to be provided with the formula to be solved by adding the respective variables, quantifiers, and clauses, and has functions to control the solving process. DepQBF was extended with assumption-based reasoning to integrate it into the framework implemented by MPIDepQBF. Apart from that, DepQBF was used out of the box without any changes.

Due to the use of assumptions, the clients are provided with the original formula ϕ to be solved in parallel only once. The master sends a set of assumptions to an idle client, which defines its subproblem to be solved, in addition to a timeout restricting the solving time. The client sends the result related to the subproblem back to the master (the result may be undefined if the solving process of the client timed out) and discards the set of assumptions. Then the master either generates a new subproblem in terms of new assumptions, or resends the previous subproblem to the client with

[5] https://www.react.uni-saarland.de/tools/caqe/index.html

an increased timeout. Information learned during a run of a client, e.g., like clauses and cubes, is not shared between the clients. However, assumption-based reasoning enables this information to be reused in different runs of the same client.

Given a PCNF $\phi := Q_1 X_1 \ldots Q_n X_n.\psi$, the master process in MPIDepQBF generates the subproblems to be solved by the clients as follows. First, the variables of each quantifier block in ϕ are sorted according to their respective number of variable occurrences. This heuristic ordering together with the quantifier ordering in the prefix of ϕ determines the order in which the variables will be assigned as assumptions to generate subproblems. Then a search tree is built in a similar way to assignment trees (cf. Fig. 1.3), which contains three types of nodes: sat, unsat, and open. Nodes of type sat and unsat represent solved subproblems whereas an open node corresponds to an unsolved subproblem and contains a variable assignment and a timeout.

Initially, the search tree is balanced and has n leaves which are of type open where n is the smallest power of 2 that is smaller than the total number of available clients. The result obtained from a client for a particular subproblem is incorporated into the search tree. For sat or unsat, the tree is simplified according to the quantifier rules in the splitting algorithm shown in Algorithm 1.2. If the result is a timeout, then the subproblem is either split further provided that additional clients are idle and hence waiting for work, or it is handed again to the same client with an increased timeout. If the tree is reduced to a single leaf node with sat or unsat then the formula is solved.

The master process is implemented in OCaml. Source code is available as part of the TOSS framework.[6] For simplifying the formula, the preprocessor bloqqer is used. MPIDepQBF is not limited to the use of DepQBF as a sequential back-end solver. In principle, any QBF solver supporting assumption-based reasoning can be integrated into MPIDepQBF. Further, the reuse of information learned locally within a run of a client has been found crucial for solving performance [46] but is not necessary for the basic workings of MPIDepQBF.

HordeQBF

The solver HordeQBF [8] is based on the massively parallel SAT-solving framework HordeSAT,[7] which integrates sequential CDCL-based SAT solvers in a portfolio style [9]. HordeSAT features hierarchical parallelism on two levels. On the top level, several instances of HordeSAT are executed in parallel and communicate with each other via MPI. These are the master processes. On the bottom level, each master starts several *core CDCL solvers* as client processes in separate threads. Thus communication within a master is implemented via the shared-memory paradigm. The clients periodically put learned clauses in

[6] http://toss.sourceforge.net/

[7] http://baldur.iti.kit.edu/hordesat/

a pool which is managed by their respective master. The pool is stored in a shared-memory region, which enables sharing of learned clauses between clients at low communication overhead. Periodically the masters exchange the learned clauses in their respective pools via MPI. This way, clauses learned by a particular client in a certain master become available to all the other clients in the different masters. The runs of the clients are diversified by providing the core solvers with different parameter settings so that the solvers operate in different parts of the search space.

HordeQBF differs from HordeSAT only in the use of a sequential QCDCL QBF solver instead of a CDCL SAT solver. The communication framework as described above is unchanged. In order to integrate a QCDCL solver into HordeQBF to be used as a core solver in the clients, the solver has to implement an API that provides functions to achieve various tasks, for example:

- import the formula in the core solver;
- diversify the run of the core solver by parameter settings;
- start the core solver;
- import/export learned clauses;
- stop the search if the formula has been solved by any core solver.

Although QCDCL solvers learn cubes in addition to clauses, the HordeSAT framework does not have to be adapted to explicitly support sharing of cubes via a dedicated API function. Instead, learned clauses and cubes are treated as sets of literals which are augmented by a special marker literal. The marker literal indicates whether the literal set is supposed to be interpreted by a client as a clause or as a cube. The master processes communicating via MPI do not distinguish between clauses or cubes but only exchange literal sets provided by the clients. Depending on certain heuristics, clients may or may not import a shared clause or cube stored in the pool of their respective master.

In principle, HordeQBF can be combined with any QCDCL QBF solver that implements the HordeSAT API. In the first release [8], the search-based solver DepQBF version 5.0, which implements a dynamic variant of *blocked clause elimination (QBCE)* for learning smaller cubes [55], was integrated into the framework.

In HordeQBF the clients check whether new learned clauses or cubes are available in the pool after a *restart*. CDCL and QCDCL solvers periodically restart by retracting the entire assignment and starting the search from scratch while keeping the learned clauses and cubes. In order to import learned clauses and cubes after a restart in DepQBF, its restart policy was modified such that it always fully retracts the assignment in a restart (cf. the original restart policy of DepQBF [56]). Learned clauses and cubes are imported, data structures are updated, and the search is resumed under the new constraints.

In order to diversify the different DepQBF instances, the master provides each solver instance with a random seed. Based on this random seed several (Q)CDCL-related parameters, such as the assignment cache [69], are randomly initialized. In consequence, the first value assigned to a decision variable is

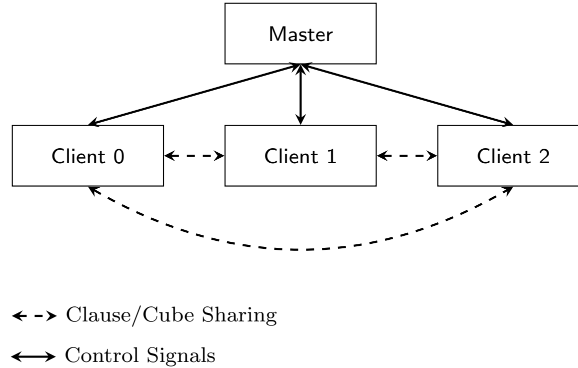

Fig. 1.5 Master-Client architecture of PAQuBE

random. Further, parameters related to variable-activity scaling (see [24]), restarting parameters, and the percentage of learned clauses and cubes to be discarded periodically are set at random. Finally, various variants of dynamic QBCE and variants of different kinds of Q-resolution to learn new constraints are randomly turned on and turned off.

Experimental results with HordeQBF on application benchmarks showed superlinear average and median speedup on a cluster with up to 1024 processing cores [8].

PAQuBE

The QBF solver PAQuBE [51, 62] is a parallel version of QuBE [33], which pioneered QCDCL solving but currently is not being further developed. QuBE implements literal watching, conflict and solution analysis, and learning as well as advanced decision heuristics. Furthermore, QuBE uses the preprocessor SqueezBF [34] which considerably improves its performance. To integrate QuBE into the parallel architecture of PAQuBE, it was extended with assumption-based reasoning (like DepQBF was extended for the integration into MPIDepQBF). For conflict analysis and backjumping, assumptions require special treatment. Furthermore, literal watching had to be modified to correctly handle clauses and cubes obtained from other clients when backtracking.

Parallelization in PAQuBE is based on MPI and a master-client architecture as shown in Figure 1.5. One dedicated master controls $n - 1$ sequential instances of QuBE. The master generates and distributes the subproblems and collects solutions using a specific variant of the guiding path method. Thereby, at any time all clients operate on subproblems rooted at variables from the same quantifier block of the given PCNF to be solved. Due to the scheduling policy, the master has to deal only with control signals but not with

shared knowledge. In consequence, the master process spends most of the time sleeping. It only has to wake up when one client is idle and a new subproblem has to be requested from another client. Hence, the master does not need its own CPU. The existence of the master process is justified by the scheduling algorithm for the distribution of subproblems. Without a master process, it would be necessary for the clients to communicate among themselves to share subproblems, thus increasing the overall communication overhead.

To solve a formula by PAQuBE, first it is read by the clients. For simplifying the formula, the preprocessor SqueezBF is applied. One client informs the master about basic formula properties such as number of variables, number of clauses, and number of quantification levels. This information is necessary for scheduling the subproblems. Then one client starts to solve the preprocessed formula as it is without any assumptions. The other clients request a subproblem from the master, who forwards their requests to the busy client. For the assignment of subproblems, the SQLS algorithm introduced with the solver QMiraXT (see below for a description of this approach) is used. SQLS is a restricted, simplified variant of the scheduling algorithm of PQSolve. The master requests a subproblem with a root variable in the current quantifier block. If the asked client does not have such a problem, another client is asked. If no client can provide a subproblem of the requested form, the master moves to the next quantification level. This will continue until either all clients are waiting for new subproblems or until a subproblem with a topmost universally (existentially) quantified variable is found unsatisfiable (satisfiable).

PAQuBE realizes an advanced knowledge-sharing mechanism of learned clauses and cubes. The clients freely communicate with each other in order to share learned clauses and cubes derived while solving their subproblems. The master process is not involved in knowledge sharing. After a fixed number of decisions the clients check whether new messages either from the master or from some other client are available. At this time, also suitable learned clauses and cubes are shared with other clients. The clients have to share the clauses and cubes learned from their run as well as receive and learn clauses and cubes derived by other clients. In addition, cubes are compressed under the assumption that different cubes share many literals from the highest quantification levels. Therefore, the literals of a cube are sorted according to the prefix order and common parts of cubes are sent only once. As this knowledge exchange leads to a significant communication overhead, multiple clause- and cube-sharing strategies are implemented. In experiments it was shown that an adaptive method yielded the overall best results. In [53] the application of machine learning is suggested to control information sharing.

PQSAT

Da Mota et al. [67] presented a parallel architecture for QBF solving. In the following, we name this approach PQSAT because it is based on the sequential

solver QSAT [70]. In contrast to most other systems, PQSAT does not require the formulas to be in PCNF. Instead it accepts arbitrarily structured formulas as input. Furthermore, the base solver QSAT used in PQSAT applies quantifier elimination rather than QCDCL. Thereby, quantified variables are successively eliminated from a given formula ϕ similarly to expansion.

PQSAT implements a parallel master-client architecture using MPI. The master reads the original QBF and splits it into several subproblems, which are distributed among the clients. For generating subproblems, the master analyses the syntactic structure of the formula in order to find subproblems which can be solved independently by the clients. For example, given the formula

$$\phi = \exists a \forall b.(((a \leftrightarrow b) \wedge (\forall c.(c \vee b))) \wedge (\exists d.(a \wedge \neg d)))$$

the subproblems $\phi_1 = \forall c.(c \vee b)$ and $\phi_2 = \exists d.(a \wedge \neg d)$ are extracted (cf. [21]). Note that variables b and a are free in ϕ_1 and ϕ_2, respectively. The task of the clients is to find propositional formulas over the free variables that are equivalent to the formulas in the subproblems by following the quantifier elimination approach implemented in QSAT [70]. For example, given a QBF $\Pi \exists x.(\psi_1 \wedge \psi_2)$ where ψ_1 does not contain any occurrence of x, the formula is rewritten to $\Pi.(\psi_1 \wedge \exists x.\psi_2)$ by minimizing the scope of $\exists x$. Then $\exists x.\psi_2$ is replaced by an equivalent formula without x. Universally quantified variables are eliminated in a similar manner. Quantifier elimination is repeated until a purely propositional formula is left. Then this propositional formula is passed to a SAT solver.

After subproblems have been assigned to the clients, the master waits for the respective results and assembles them in order to get the result of the full problem. As the subproblems may contain free variables, the clients must return an equivalent formula over these free variables without any quantifiers. The clients themselves may split their given subproblems into further subproblems if the given subproblem appears to be too difficult according to some syntactic measure of difficulty. If a client decides to split a subproblem, then it employs semantic splitting based on assignments to the free variables. The set of new subproblems is passed to the master node, who distributes them to other idle clients.

PQSolve

One of the first parallel QBF solvers was PQSolve, which was published in the year 2000 [28]. At that time, QBF-solving technology in general still was in its infancy. For example, neither learning as used in QCDCL-based QBF solvers nor expansion-based solving had been presented. Although PQSolve naturally lacks many techniques that are standard in modern solvers, it can be seen as a milestone in parallel QBF solving. PQSolve relies on QSolve as the base solver, which implements the DPLL algorithm for QBF with several then

state-of-the-art heuristics and pruning techniques such as quantifier inversion, trivial truth, and trivial falsity [18, 74]. Thus PQSolve is an early distributed realization of DPLL for QBF.

The motivation for parallelizing QSolve stems from the common view of QBF solving as a two-person zero-sum game with complete information (cf. [76]). Thereby, the *universal player* assigns the universally quantified variables of a given QBF with the aim to falsify the formula, whereas the *existential player* assigns the existentially quantified variables in order to satisfy it. For the development of PQSolve, its authors applied techniques successfully used in parallel chess programs.

PQSolve implements a master-client architecture based on MPI where the role of master and client processes may change dynamically depending on the scheduling of subproblems and on the progress of the search. Furthermore, there may be more than one master process. This dynamic architecture of PQSolve is different from many other parallel QBF solvers and complicates the checking of termination conditions. To obtain a simpler design, solvers such as QMiraXT and PAQuBE implement a restricted variant of PQSolve's architecture and scheduling based on the SQLS algorithm.

PQSolve takes formulas in PCNF as input and works as follows: first one process is assigned to solve the input formula. All other processes are idle. If a process Q is idle then it sends a request for work to a random process P which is not idle. If the contacted busy process P has an unexplored part in its current search tree then it sends the respective formula to the requesting process Q similarly to the guiding path method. This way, P becomes the master of the client Q. The requesting client process Q now solves the formula and sends the result back to the master P. Then Q becomes idle again and the master-client relationship between P and Q is released. Process P incorporates the result into its search tree. If P has another open subproblem then it communicates that subproblem to the idle process Q. Otherwise, a request for work is sent to a random busy process. It may happen that a client's work on a subproblem becomes obsolete because of some pruning techniques applied in the master. In this case, the master informs the client to stop solving the respective subproblem.

Every process in PQSolve applies tests for trivial truth and trivial falsity. For the trivial truth check, only the existentially quantified variables are considered and all literals of universal variables are discarded from the PCNF. If the resulting propositional formula is satisfiable, then also the original PCNF is satisfiable. For the trivial falsity check, all variables are assumed to be existentially quantified. If the resulting propositional formula is unsatisfiable, then also the original PCNF is unsatisfiable. Trivial truth and falsity checks are simply realized with a SAT solver and can be done at any time during the search.

The subproblem handed over to a different process in PQSolve must be large enough to justify the communication overhead. The selection and scheduling of subproblems work as follows. Let $\{l_0, \ldots, l_m\}$ be the current assignment

such that l_i was assigned before l_j if $i < j$. When receiving a request from another process, then the formula under assignment $\{l_0, \dots, \bar{l}_i\}$ is passed to the other process such that $3 * |N(x_i) - P(x_i)| + i$ is minimal where $\mathsf{var}(l_i) = x_i$ and $P(x_i)$ is the number of positive occurrences of x_i and $N(x_i)$ is the number of negative occurrences of x_i.

To increase parallel efficiency, PQSolve implements *Helpful Master Scheduling*. A master process that has passed on a subproblem to a client has to wait for the result and thus stays idle after it has solved its own subproblem. In that case the master itself sends a request to the client, which in turn provides a subproblem (of its current one) to share the work.

To avoid irrelevant work, *Young Brothers Wait Scheduling* is applied. This approach tries to deal with the problem that when solving a formula under a certain assignment of some variable x, it is often not necessary to solve the formula under the dual assignment of x. In a parallel setting, situations of this kind result in a waste of work. Therefore, blocks of variables are considered. Only after the leftmost leaves of the subtrees obtained by setting the variables in a block have been fully evaluated are the subformulas related to the other subtrees passed to other processes.

PQUABS

The solver PQUABS [83] extends the sequential solver quabs [84], which processes formulas in prenex negation normal form (prenex NNF), by allowing input formulas to be in non-prenex NNF. That is, PQUABS is able to handle formulas with a tree-shaped quantifier structure in contrast to the linear quantifier structure of formulas in prenex NNF. For each maximal consecutive block of quantifiers of the same type, PQUABS builds a propositional abstraction of the input formula in a way that is similar to the approach implemented in caqe [73] and its parallel variant pcaqe. Thereby, the evaluation of a given QBF is broken down to evaluating a set of propositional abstractions. The abstraction of a quantifier block is linked to the abstractions of adjacent quantifier blocks in the syntactic structure of the formula via so-called interface literals. The interface literals express quantifier dependencies resulting from the ordering of quantifier blocks. The satisfiability of a subformula is communicated via assignments to the interface literals. There are two types of interface literals: one type to represent the assignments made by abstractions of outer quantifier blocks, and the other type to represent the assignments made by abstractions of inner quantifier blocks. A counterexample-guided abstraction refinement loop (CEGAR) is employed based on SAT solving to generate refined abstractions. Additionally, PQUABS analyses the quantifier structure of the given formula to avoid the use of interface literals whenever a subformula appears in the scope of only one quantifier block.

QMiraXT

The solver QMiraXT [52, 77] implements QCDCL combined with preprocessing. Unlike the other parallel QCDCL solvers (see Table 1.1), knowledge sharing is based on shared memory (see Figure 1.6) rather than message passing by MPI. QMiraXT is an extension of the parallel SAT solver MiraXT. While MiraXT and QMiraXT share a common architecture, the reasoning mechanisms of QMiraXT are adapted to QBF.

QMiraXT implements a decision heuristics similar to VSIDS [66], but takes the different quantification levels of the variables into account. That is, all variables of the current level have to be set before a variable of the next level is selected, similarly to QBF semantics. Two counters are used to keep track of positive and negative variable occurrences in the formula. When conflict clauses are added, these counters are increased. Further, they are periodically decreased to amplify the influence of more recent conflict clauses. From a set of existentially quantified variables, the variable that satisfies the largest number of clauses is chosen. A universally quantified variable is selected and assigned so that the number of implications by unit clauses that would result from the respective assignment is maximized.

QMiraXT eliminates unused variables and pure literals and performs substitution of equivalent literals. Then the complete solver Quantor [12] is applied as preprocessor. Quantor implements bounded variable elimination and universal variable expansion. Those techniques are applied until the formula reduces to a propositional formula. As the memory consumption of Quantor is not restricted, QMiraXT sets a memory limit (128 MB is reported in [52]) as well as a time limit of five seconds. Then the remaining QBF formula is processed in QCDCL style. This way, Quantor is applied in an incomplete manner as a preprocessor, what is very similar to the idea behind the preprocessor bloqqer.

The shared clause database (SCD) of QMiraXT contains every clause that is currently used by a client thread. Cubes are not stored because it was found [52] that in general they are too large, and storing them would slow down the performance of the solver. A clause is contained only once in the SCD and is marked as read-only. After a clause has been generated and added to the SCD it is available to all threads via shared-memory accesses. That is, unlike MPI-based communication as implemented in other parallel solvers, explicit exchange of shared clauses via messages is not required.

Clauses stored in the SCD may reside at any position in memory. To optimize memory accesses made by the threads, each thread maintains a *watched-literal reference list (WLRL)*. For every clause, the WLRL allows a thread to store two watched literals and an existentially quantified *cache literal* in its local memory. It has been shown that this caching policy optimizes memory accesses made by the threads.

QMiraXT has no controlling master process. Instead there is a *Master Control Object* (MCO), which coordinates the communication between the threads. The MCO is never directly involved in the communication. It stores messages

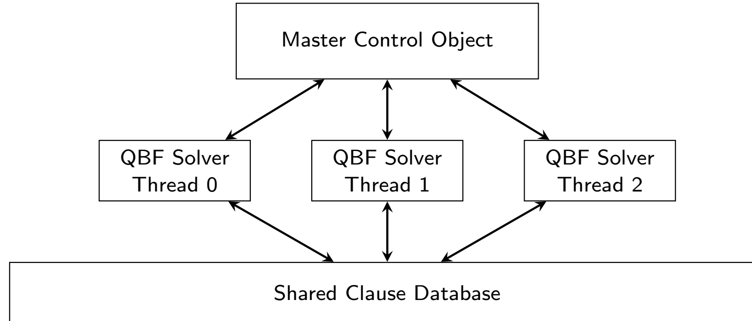

Fig. 1.6 Architecture of the solver QMiraXT [52]

on global events, for example, that the formula has been solved. The most important task of the MCO is the generation of subproblems. Subproblems are generated by the guiding path method like in the solvers MPIDepQBF, PAQuBE, and PQSolve. To this end, the MCO provides the two functions `donateDecisionStack()` and `getDecisionStack()`, which both enforce the use of locks. Function `donateDecisionStack()` splits the *decision stack* of the current thread into two different decision stacks and provides another thread with one of them. The decision stack contains the decision variables in the ordering they were assigned in QCDCL. Function `getDecisionStack()` implements *single quantification level scheduling (SQLS)* [77]. SQLS is a restricted, simplified variant of the scheduling algorithm of PQSolve and is also employed by the solver PAQuBE. The splitting of the search space is done by the clients. Clients are allowed to split the search space on one quantification level. If no subproblems are available anymore, the threads block until either new subproblems have been provided by another thread or until all other threads terminate. If only one thread is running and all other decisions have been considered at the current decision level, it may use variables from the next quantification level. In this way, the clients manage the subproblem generation by themselves and no complicated management infrastructure is needed.

1.6 Challenges and Potential of Parallel QBF Solving

In the past and recent QBF solver landscape, the majority of the presented tools focus on sequential solving approaches. Thus the potential of modern computer architectures is currently not fully leveraged. In the following, we discuss challenges and opportunities that arise in the context of parallel QBF solving.

Preprocessing

In the context of sequential QBF solving, preprocessing has been shown to be substantially valuable to many state-of-the-art solvers [60]. All parallel approaches either perform some simplifications before solving, exploit the power of sequential preprocessors such as bloqqer, SqueezBF, or HQSpre [87], or apply the complete solver Quantor in a resource bounded way. In general, the goal of preprocessing is to simplify the input formula such that it becomes easier to solve. At the moment, however, no special parallel preprocessing techniques are applied. It would be a natural approach to apply expensive sequential preprocessing techniques in parallel. To this end, however, it has to be investigated whether preprocessing techniques that have been found beneficial in the context of sequential solving are also beneficial to parallel solving to the same extent. Furthermore, it may be necessary to tune sequential preprocessing techniques to parallel settings. For example, whereas in sequential solving the elimination of both variables and clauses from a formula is crucial, in parallel solving it may be more important to emphasize the removal of variables. Search space splitting is carried out based on the set of variables. Hence eliminating variables reduces the size of the search space and thus might simplify search space splitting.

Learning and Knowledge-Sharing Heuristics

All parallel solvers which support knowledge sharing (see Table 1.1) are based on QCDCL. In QCDCL, new learned clauses and cubes are derived using the Q-resolution calculus (Definition 1). The learned clauses can be shared with other threads or processes in order to prune the search space and thus speed up the overall search.

To limit the communication overhead that may result from sharing, suitable heuristics must be applied in order to select the clauses and cubes to be shared. For example, in parallel SAT solving, typical clause selection metrics are the length of a clause or the involvement of literals in conflicts.

While the quantifier structure of QBFs results in several restrictions that potentially limit the effectiveness of parallel solving techniques in general, at the same time it gives rise to additional selection criteria. Possible criteria are the number of universal (existential) literals in a clause (cube), or the quantification levels of literals of a clause (cube).

Effective selection criteria are particularly important when it comes to sharing learned cubes. In QCDCL learned cubes are first derived by rule *cu-init* of the Q-resolution calculus. Cubes derived this way tend to be large since their derivation relies on assignments that satisfy all clauses of the given PCNF. Due to the size of cubes, it may be costly to share large numbers of cubes. Furthermore, large cubes tend to prune only small parts of the search

space. We see a lot of potential in the development of useful heuristics to decide on the benefit of sharing knowledge.

In general, cube learning in QCDCL may be a bottleneck also in sequential QBF solving. To mitigate the weaknesses of deriving only large cubes by rule *cu-init*, the Q-resolution calculus has been extended by additional axioms [59]. Derivations made by these additional axioms rely on the application of oracles to check the satisfiability of QBFs that arise during the solving process. In this respect, oracles implement resource-bounded procedures for QBF satisfiability checking. Cubes derived by the additional axioms are potentially smaller than cubes derived by rule *cu-init* in the traditional way. In parallel QBF solving based on QCDCL, there is considerable potential in parallelizing the calls of several oracles, which might implement orthogonal or incomplete solving techniques, for example. Since the cubes derived by such parallel oracle calls tend to be smaller than cubes derived by rule *cu-init*, sharing these cubes with other threads or processes in the solver would result in smaller communication overhead and better pruning of the search space.

Incremental Solving

An *incremental* QBF solver based on QCDCL [58, 61, 65] allows us to solve sequences $S := \langle \phi_0, \ldots, \phi_n \rangle$ of related PCNFs ϕ_i. Each PCNF ϕ_{i+1} is obtained from the previous PCNF ϕ_i by adding or deleting clauses, variables, or quantifiers. When solving a PCNF ϕ_i in S in an incremental way, the solver does not start from scratch. Instead, clauses and cubes learned when solving ϕ_i potentially can be kept and reused when solving the next PCNF ϕ_{i+1}. This way, the PCNFs ϕ_i might be solved faster than if each ϕ_i was solved independently and non-incrementally. For incremental solving, the solver must provide an API so that the same solver instance can be used to solve the PCNFs in S.

We are not aware of any approaches to parallelize incremental QBF solving. Hence the potential positive effects of combining the benefits of incremental and parallel solving are currently not leveraged. It might be possible to apply approaches from incremental and parallel SAT solving [86] also to QBF.

Expansion-Based Solving

Currently most parallel solvers implement search-based solving by QCDCL (see Table 1.1 above). However, recently expansion-based solving [5, 12] in combination with CEGAR [42, 43] has been shown to be powerful in solving many practically relevant classes of formulas.

Since expansion is orthogonal to QCDCL regarding proof complexity [11, 43], there is considerable potential in parallelizing solvers that employ CEGAR-based expansion. However, it has not been deeply investigated how to leverage

the power of CEGAR approaches in parallel solving. For example, different processes could work on different abstractions of a formula at the same time and then share or synchronize counterexamples that they have found with respect to the different abstractions.

Duality-Aware Reasoning

In the context of QBF solving it is well known that reasoning on a propositional CNF introduces a bias towards the search for conflicts. A CNF is easily falsified by an assignment that falsifies at least one clause. Based on such falsifying assignments, in QCDCL learned clauses can be derived by rule *cl-init* of the Q-resolution calculus. Compared to falsifying assignments, it is more difficult to satisfy a CNF as all the clauses must be satisfied. Therefore, for the search for solutions, a formula in disjunctive normal form (DNF), i.e., a disjunction of cubes, would be better suited. A DNF is dual to a CNF in the sense that a DNF can be satisfied easily by satisfying at least one of its cubes.

To benefit from properties of both CNFs and DNFs, approaches have been presented that reason on a CNF and on a DNF representation of the given QBF at the same time. This way, propagations are performed on the CNF and on the DNF (e.g., [37, 49, 89]). However, so far these approaches have been realized systematically only in a sequential manner. The solver par-pd-depqbf is based on the observation of Van Gelder [30] who proposes to solve a formula in CNF as well as in DNF by calling two separate solver instances in parallel. However, in this approach there is neither communication nor knowledge sharing between the solver instances.

Proof Generation

The generation of proofs becomes more and more important for the practical applicability of QBF solvers. Proofs serve two purposes: on the one hand, they allow for the independent validation of the correctness of a solver's result by an efficient checker, and on the other hand they allow the extraction of *Skolem* and *Herbrand functions*. Skolem functions represent a *strategy* for the assignment of existential variables if a formula is satisfiable. Likewise, Herbrand functions represent a strategy for selecting the assignments of universal variables in unsatisfiable formulas (see also the informal presentation of these functions by means of the example in Fig. 1.1 in Section 1.1).

Strategies are crucial for practical applications of QBF solvers. For example, given a PCNF ϕ which models an instance of some problem to be solved, a solution to the problem instance can be computed from a strategy for ϕ.

Skolem and Herbrand functions can be efficiently extracted from *Q-resolution proofs* as produced by sequential QCDCL solvers [6]. However, in parallel QBF solving, currently none of the presented approaches supports

the generation of proofs or strategies in terms of Skolem and Herbrand functions, respectively. However, for parallel solvers based on QCDCL a potential approach to proof generation would be to combine the respective proofs of the subproblems that have been solved by the different threads or processors. To this end, ideas from proof generation in parallel SAT solving [39] may also be applicable.

Testing and Debugging

One of the major challenges in developing a sequential QBF solver is a stable implementation, i.e., an implementation which does not crash and which returns correct results. In general, implementations of QBF solvers are more complex than implementations of SAT solvers due to the complexity of handling nested quantifiers that is present in QBFs. Furthermore, in order to achieve good solving performance, it is necessary to equip QBF solvers with advanced data structures and optimizations to prune the search space. At the same time, these optimizations may hinder the efficient implementation of advanced features such as proof generation and incremental solving.

For sequential solving, effective approaches to testing and debugging of solvers [16] exist. First, *fuzz testing* has proven itself to be powerful for finding problematic corner cases and conceptual errors in an implementation. A fuzz test generates random formulas according to predefined random models such that the formulas are not too hard to solve. The goal is to achieve a high testing throughput together with a uniform distribution of satisfiable and unsatisfiable instances.

Second, *delta debugging* is used to automatically simplify large formulas on which a solver exhibits incorrect behavior. To this end, clauses are successively removed from the formula and literals are removed from clauses so that the incorrect behavior of the solver is preserved. In the end, the result of delta debugging is a formula which is reasonably small so that the run of the solver can be inspected manually using traditional debugging techniques.

Third, *model-based testers* [1, 2] have been found particularly useful in testing the behavior of incremental solvers via the API provided for incremental use. While in fuzz testing and delta debugging solvers are considered as black boxes, a model-based testing environment comes with a tighter integration of the solver. Sequences of function calls of the solver's API are automatically generated and replayed in order to test solver behavior on the sequence. This approach also allows us to replay entire solver runs where certain bugs were triggered.

It is well known that testing and debugging of parallel solvers is far more complex than for sequential solvers. In parallel QBF solving, this problem is made worse by the higher complexity that is intrinsic to QBF solvers, compared to SAT solvers. For the development of robust parallel QBF solvers,

it may be useful to combine the generation of proofs and strategies outlined above with approaches to automated testing and debugging.

1.7 Conclusion

Already in the very early years of QBF solving attempts were made to exploit the full computational power of modern computer architectures, ranging from multicore processors to huge clusters as found in modern cloud-based systems. However, compared to the advancements made in sequential QBF solving, a lot of the potential of parallelizing QBF solving has still not been exploited.

We have reviewed and classified parallel approaches to QBF solving that either were published in the literature or that participated in the parallel track of the QBF competition QBFEVAL'16 held in 2016. Overall, we identified 11 approaches; 10 of them are implemented. The implementations of five systems are publicly available. Unfortunately, not all of the QBFEVAL'16 participants are among those solvers. As half of the systems are not available (anymore), we did not carry out an empirical evaluation. The parallel track of QBFEVAL'16 was not competitive due to the small number of participating systems. However, it is still remarkable that the track could be carried out, because in the previous editions of QBFEVAL it had to be canceled. This fact might be a first indicator of an upwards trend in parallel QBF solving. Given the high computational complexity of QBF solving in general, the large variety of sequential solvers and the power of modern computer architectures, we see considerable potential to speed up QBF solving by parallel approaches.

References

1. Cyrille Artho, Armin Biere, and Martina Seidl. Model-Based Testing for Verification Back-Ends. In Margus Veanes and Luca Viganò, editors, *Proc. of the 7th Int. Conference on Tests and Proofs (TAP 2017)*, volume 7942 of *LNCS*, pages 39–55. Springer, 2013.
2. Cyrille Artho, Martina Seidl, Quentin Gros, Eun-Hye Choi, Takashi Kitamura, Akira Mori, Rudolf Ramler, and Yoriyuki Yamagata. Model-Based Testing of Stateful APIs with Modbat. In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *Proc. of the 30th Int. Conference on Automated Software Engineering (ASE 2015)*, pages 858–863. IEEE Computer Society, 2015.
3. Bengt Aspvall, Christos Levcopoulos, Andrzej Lingas, and Robert Storlind. On 2-QBF Truth Testing in Parallel. *Information Processing Letters*, 57(2):89–93, 1996.
4. Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.
5. Abdelwaheb Ayari and David A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In Mark Aagaard and John W. O'Leary, ed-

itors, *Proc. of the 4th Int. Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.
6. Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012.
7. Valeriy Balabanov, Jie-Hong Roland Jiang, Mikolás Janota, and Magdalena Widl. Efficient Extraction of QBF (Counter)models from Long-Distance Resolution Proofs. In Blai Bonet and Sven Koenig, editors, *Proc. of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 3694–3701. AAAI Press, 2015.
8. Tomas Balyo and Florian Lonsing. HordeQBF: A Modular and Massively Parallel QBF Solver. In Nadia Creignou and Daniel Le Berre, editors, *Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *LNCS*, pages 531–538. Springer, 2016.
9. Tomas Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A Massively Parallel Portfolio SAT Solver. In Marijn Heule and Sean Weaver, editors, *Proc. of the 18th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2015)*, volume 9340 of *LNCS*, pages 156–172. Springer, 2015.
10. Marco Benedetti and Hratch Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *Journal on Satisfiability, Boolean Modeling and Computation*, 5(1-4):133–191, 2008.
11. Olaf Beyersdorff, Leroy Chew, and Mikolás Janota. Proof Complexity of Resolution-based QBF Calculi. In Ernst W. Mayr and Nicolas Ollinger, editors, *Proc. of the 32nd Int. Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *LIPIcs*, pages 76–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
12. Armin Biere. Resolve and Expand. In Holger H. Hoos and David G. Mitchell, editors, *Proc. of the 7th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *LNCS*, pages 59–70. Springer, 2004.
13. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
14. Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-Based Synthesis Methods for Safety Specs. In Kenneth L. McMillan and Xavier Rival, editors, *Proc. of the 15th Int. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014)*, volume 8318 of *LNCS*, pages 1–20. Springer, 2014.
15. Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Solving QBF instances with nested SAT solvers. In Adnan Darwiche, editor, *Proc. of the 2016 AAAI Workshop Beyond NP*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.
16. Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Proc. of the 13th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.
17. Uwe Bubeck and Hans Kleine Büning. Bounded Universal Expansion for Preprocessing QBF. In *Proc. of the 10th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *LNCS*, pages 244–257. Springer, 2007.
18. Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In Jack Mostow and Chuck Rich, editors, *Proc. of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)*, pages 262–267. AAAI Press / The MIT Press, 1998.
19. Koen Claessen, Niklas Eén, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson. SAT-Solving in Practice, with a Tutorial Example from Supervisory Control. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.
20. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

21. Benoit Da Mota. *Quantified Boolean formulae: formal processings and parallel computations.* Theses, Université d'Angers, December 2010.
22. Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
23. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Proc. of the 8th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
24. Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Proc. of the 9th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
25. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
26. Uwe Egly, Martin Kronegger, Florian Lonsing, and Andreas Pfandler. Conformant planning as a case study of incremental QBF solving. *Ann. Math. Artif. Intell.*, 80(1):21–45, 2017.
27. Wolfgang Faber and Francesco Ricca. Solving hard ASP programs efficiently. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proc. of the 8th Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, volume 3662 of *LNCS*, pages 240–252. Springer, 2005.
28. Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In Henry A. Kautz and Bruce W. Porter, editors, *Proc. of the 17th Nat. Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence (AAA/IAAI 2000)*, pages 285–290. AAAI Press / The MIT Press, 2000.
29. Allen Van Gelder. Contributions to the theory of practical quantified boolean formula solving. In Michela Milano, editor, *Proc. of the 18th Int. Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *LNCS*, pages 647–663. Springer, 2012.
30. Allen Van Gelder. Primal and Dual Encoding from Applications into Quantified Boolean Formulas. In Christian Schulte, editor, *Proc. of the 19th Int. Conference on Principles and Practice of Constraint Programming (CP 2013)*, volume 8124 of *LNCS*, pages 694–707. Springer, 2013.
31. Ian P. Gent, Enrico Giunchiglia, Massimo Narizzano, Andrew G. D. Rowley, and Armando Tacchella. Watched data structures for QBF solvers. In Enrico Giunchiglia and Armando Tacchella, editors, *Proc. of the 6th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 25–36. Springer, 2003.
32. Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. Reasoning with quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 761–780. IOS Press, 2009.
33. Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. QuBE7.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):83–88, 2010.
34. Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In Ofer Strichman and Stefan Szeider, editors, *Proc. of the 13th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *LNCS*, pages 85–98. Springer, 2010.
35. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res. (JAIR)*, 26:371–416, 2006.
36. Alexandra Goultiaeva and Fahiem Bacchus. Recovering and Utilizing Partial Duality in QBF. In Matti Järvisalo and Allen Van Gelder, editors, *Proc. of the 16th Int.*

Conference on Theory and Applications of Satisfiability Testing (SAT 2013), volume 7962 of *LNCS*, pages 83–99. Springer, 2013.
37. Alexandra Goultiaeva, Martina Seidl, and Armin Biere. Bridging the gap between dual propagation and CNF-based QBF solving. In Enrico Macii, editor, *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE 2013)*, pages 811–814. EDA Consortium / ACM DL, 2013.
38. Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause Elimination for SAT and QSAT. *J. Artif. Intell. Res. (JAIR)*, 53:127–168, 2015.
39. Marijn J. H. Heule and Armin Biere. Compositional Propositional Proofs. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Proc. of the 20th Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning(LPAR-20)*, volume 9450 of *LNCS*, pages 444–459. Springer, 2015.
40. Tamir Heyman, Dan Smith, Yogesh Mahajan, Lance Leong, and Husam Abu-Haimed. Dominant Controllability Check Using QBF-Solver and Netlist Optimizer. In Carsten Sinz and Uwe Egly, editors, *Proc. of the 17th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*, volume 8561 of *LNCS*, pages 227–242. Springer, 2014.
41. Mikolás Janota, Charles Jordan, Will Klieber, Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF Gallery 2014: The QBF Competition at the FLoC Olympic Games. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:187–206, 2016.
42. Mikolás Janota, William Klieber, Joao Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.
43. Mikolás Janota and Joao Marques-Silva. Expansion-based QBF solving versus Q-resolution. *Theor. Comput. Sci.*, 577:25–42, 2015.
44. Mikolás Janota and Joao Marques-Silva. Solving QBF by Clause Selection. In Qiang Yang and Michael Wooldridge, editors, *Proc. of the 24th Int. Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 325–331. AAAI Press, 2015.
45. Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing Rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proc. of the 6th Int. Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.
46. Charles Jordan, Lukasz Kaiser, Florian Lonsing, and Martina Seidl. MPIDepQBF: Towards Parallel QBF Solving without Knowledge Sharing. In Carsten Sinz and Uwe Egly, editors, *Proc. of the 17th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*, volume 8561 of *LNCS*, pages 430–437. Springer, 2014.
47. Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 735–760. IOS Press, 2009.
48. Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1):12–18, 1995.
49. William Klieber, Samir Sapra, Sicun Gao, and Edmund M. Clarke. A non-prenex, non-clausal QBF solver with game-state learning. In Ofer Strichman and Stefan Szeider, editors, *Proc. of the 13th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *LNCS*, pages 128–142. Springer, 2010.
50. Reinhold Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In Uwe Egly and Christian G. Fermüller, editors, *Proc. of the Int. Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2002)*, volume 2381 of *LNCS*, pages 160–175. Springer, 2002.
51. Matthew Lewis, Tobias Schubert, Bernd Becker, Paolo Marin, Massimo Narizzano, and Enrico Giunchiglia. Parallel QBF Solving with Advanced Knowledge Sharing. *Fundamenta Informaticae*, 107(2-3):139–166, 2011.

52. Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. QmiraXT - A Multithreaded QBF Solver. In Carsten Gremzow and Nico Moser, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 7–16. Universitätsbibliothek Berlin, Germany, 2009.
53. Tao Li and Nan-feng Xiao. Parallel solving model for quantified boolean formula based on machine learning. *Journal of Central South University*, 20(11):3156–3165, 2013.
54. Paolo Liberatore. Redundancy in logic I: CNF propositional formulae. *Artif. Intell.*, 163(2):203–232, 2005.
55. Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing Search-Based QBF Solving by Dynamic Blocked Clause Elimination. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Proc. of the 20th Int. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015)*, volume 9450 of *LNCS*, pages 418–433. Springer, 2015.
56. Florian Lonsing and Armin Biere. DepQBF: A Dependency-Aware QBF Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):71–76, 2010.
57. Florian Lonsing and Armin Biere. Integrating dependency schemes in search-based QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Proc. of the 13th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *LNCS*, pages 158–171. Springer, 2010.
58. Florian Lonsing and Uwe Egly. Incremental QBF Solving. In Barry O'Sullivan, editor, *Proc. of the 20th Int. Conference on Principles and Practice of Constraint Programming (CP 2014)*, volume 8656 of *LNCS*, pages 514–530. Springer, 2014.
59. Florian Lonsing, Uwe Egly, and Martina Seidl. Q-Resolution with Generalized Axioms. In Nadia Creignou and Daniel Le Berre, editors, *Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *LNCS*, pages 435–452. Springer, 2016.
60. Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF Gallery: Behind the scenes. *Artif. Intell.*, 237:92–114, 2016.
61. Paolo Marin, Christian Miller, Matthew D. T. Lewis, and Bernd Becker. Verification of partial designs using incremental QBF solving. In Wolfgang Rosenstiel and Lothar Thiele, editors, *Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE 2012)*, pages 623–628. IEEE, 2012.
62. Paolo Marin, Massimo Narizzano, Enrico Giunchiglia, Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Comparison of knowledge sharing strategies in a parallel QBF solver. In *Proc. of the Int. Conference on High Performance Computing & Simulation (HPCS 2009)*, pages 161–167. IEEE, 2009.
63. Paolo Marin, Massimo Narizzano, Luca Pulina, Armando Tacchella, and Enrico Giunchiglia. Twelve Years of QBF Evaluations: QSAT Is PSPACE-Hard and It Shows. *Fundam. Inform.*, 149(1-2):133–158, 2016.
64. Albert R. Meyer and Larry J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *13th Annual Symposium on Switching and Automata Theory*, pages 125–129. IEEE Computer Society, 1972.
65. Christian Miller, Paolo Marin, and Bernd Becker. Verification of partial designs using incremental QBF. *AI Commun.*, 28(2):283–307, 2015.
66. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM, 2001.
67. Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. A new parallel architecture for QBF tools. In *Proc. of the Int. Conference on High Performance Computing and Simulation (HPCS 2010)*, pages 324–330. IEEE, 2010.
68. Alexander Nadel and Vadim Ryvchin. Efficient SAT Solving under Assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Proc. of the 15th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *LNCS*, pages 242–255. Springer, 2012.

69. Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Proc. of the 10th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *LNCS*, pages 294–299. Springer, 2007.
70. David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.
71. Luca Pulina. The Ninth QBF Solvers Evaluation - Preliminary Report. In *Proc. of the 4th Int. Workshop on Quantified Boolean Formulas (QBF 2016)*, volume 1719, pages 1–13. CEUR Workshop Proceedings, 2016.
72. Markus N. Rabe and Sanjit A. Seshia. Incremental Determinization. In Nadia Creignou and Daniel Le Berre, editors, *Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *LNCS*, pages 375–392. Springer, 2016.
73. Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In Roope Kaivola and Thomas Wahl, editors, *Proc. of the Int. Conference on Formal Methods in Computer-Aided Design (FMCAD 2015)*, pages 136–143. IEEE, 2015.
74. Jussi Rintanen. Improvements to the evaluation of quantified boolean formulae. In Thomas Dean, editor, *Proc. of the 16th Int. Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 1192–1197. Morgan Kaufmann, 1999.
75. Jussi Rintanen. Asymptotically Optimal Encodings of Conformant Planning in QBF. In *Proc. of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 1045–1050. AAAI Press, 2007.
76. Thomas J. Schaefer. On the Complexity of Some Two-Person Perfect-Information Games. *J. Comput. Syst. Sci.*, 16(2):185–225, 1978.
77. Tobias Schubert, Matthew D. T. Lewis, and Bernd Becker. Pamiraxt: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.
78. João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.
79. João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proc. of the Int. Conference on Computer-Aided Design (ICCAD 1996)*, pages 220–227, 1996.
80. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
81. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proc. of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 1–9, New York, NY, USA, 1973. ACM.
82. Larry J. Stockmeyer. The Polynomial-Time Hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.
83. Leander Tentrup. Non-prenex QBF Solving Using Abstraction. In *In Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *LNCS*, pages 393–401. Springer, 2016.
84. Leander Tentrup. Solving QBF by abstraction. *CoRR*, abs/1604.06752, 2016.
85. Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.
86. Siert Wieringa and Keijo Heljanko. Asynchronous Multi-core Incremental SAT Solving. In Nir Piterman and Scott A. Smolka, editors, *Proc. of the 19th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *LNCS*, pages 139–153. Springer, 2013.
87. Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker. HQSpre - An Effective Preprocessor for QBF and DQBF. In *Proc. of the 23rd Int. Conference on Tools*

and Algorithms for the Construction and Analysis of Systems (TACAS 2017), volume 10205 of *LNCS*, pages 373–390. Springer, 2017.
88. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *J. Symb. Comput.*, 21(4):543–560, 1996.
89. Lintao Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *Proc. of the 21st Nat. Conference on Artificial Intelligence and the 8th Innov. Applications of Artificial Intelligence Conference (AAAI/IAAI 2006)*, pages 143–150. AAAI Press, 2006.
90. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In Rolf Ernst, editor, *Proc. of the Int. Conference on Computer-Aided Design (ICCAD 2001)*, pages 279–285. IEEE, 2001.
91. Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean Satisfiability solver. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proc. of the Int. Conference on Computer-Aided Design (ICCAD 2002)*, pages 442–449. ACM / IEEE Computer Society, 2002.

Index

QuBE, 22

antecedent clause, 10
asserting clause, 10
assignment cache, 21
assignment tree, 3, 6, 8, 10, 13, 20
assumption-based reasoning, 22

backjumping, 22
bloqqer, 18, 20, 27, 29
bounded expansion, 4

caqe, 16–19, 26
CDCL, 3, 4, 8, 9
CEGAR, 26, 30
clause learning, 8, 29
conflict analysis, 22
conflict clause, 27
cube learning, 29

delta debugging, 32
DepQBF, 16–19, 21, 22
DPLL, 3, 8, 24, 25
duality-aware reasoning, 4, 31

existential reduction, 7, 11
expansion-based QBF solving, 4, 12, 30
expansion-based solving, 24

fuzz testing, 32

guiding path, 14, 15, 22, 28

helpful master scheduling, 26
Herbrand function, 31
hiqqer, 18
hiqqerfork, 12, 17, 18
HordeQBF, 12, 16, 17, 20–22
HordeSAT, 12, 20, 21
hqspre, 29

implication graph, 17
information sharing, 17, 29
inprocessing, 4, 17

knowledge sharing, 21, 23

learning, 22
literal watching, 22

master control object, 27
Minisat, 19
model-based testing, 32
MPI, 22
MPIDepQBF, 15, 17, 19, 20, 22, 28

NP-completeness, 2

PAQuBE, 15, 17, 22, 23, 25, 28
par-pd-depqbf, 12, 17, 18, 31
partial assignment, 6
pcaqe, 16–19, 26
PCNF, *see* prenex conjunctive normal form
Picosat, 19
pivot variable, 7
polynomial hierarchy, 2
portfolio solving, 11–12, 17, 18, 20
PQSAT, 16, 17, 23, 24
PQSolve, 15, 17, 23–26, 28
PQUABS, 16, 17, 19, 26
prenex conjunctive normal form, 5
prenex negation normal form, 26
preprocessing, 4, 17, 22, 27, 29

propositional satisfiability, 1–2
PSPACE, 17
PSPACE-completeness, 3

Q-resolution, 6–9, 11, 15
  proof, 7, 31
Q-resolution calculus, 7
Q2CNF, 17
QBCP, 9, 10
QBF, 2–33
  assignment, 6
  assumption-based reasoning, 19
  blocked clause elimination, 21
  clause, 5
  clause learning, 7, 9, 10
  closed formula, 5
  conflict, 3, 10
  conjunctive normal form, 5
  countermodel, 2, 6
  cube, 5
  cube learning, 7, 9, 10
  decision making, 8–10
  disjunctive normal form, 5
  existential reduction, 7
  expansion-based solving, 4, 16, 30
  free formula, 5
  incremental solving, 16, 30
  inprocessing, 16
  knowledge sharing, 14, 16
  learning, 8, 16
  matrix, 5
  model, 2, 6
  negation normal form, 6
  preprocessing, 4, 16, 20
  pure literal, 9, 27
  quantifier scope, 5
  restart, 21
  satisfiability-equivalent, 6
  search-based solving, 3, 8–11, 30
  semantics, 6
    existential player, 25
    game, 25
    universal player, 25
  solution, 4, 10
  strategy, 31
  syntax, 5
  unit clause, 11
  unit literal, 9–11
  unit propagation, 8
  variable assignment, 26
QBFEVAL, 4, 17, 18, 33
QCDCL, 3, 4, 7–13, 15–17, 19–21, 27, 31
QCIR, 12
QMiraXT, 15–17, 23, 25, 27, 28
QSAT, 2, 16, 17, 24
QSolve, 16, 17, 24, 25
quabs, 16, 17, 26
quantified Boolean formulas, *see* QBF
quantifier elimination, 24
quantifier inversion, 25
Quantor, 27, 29
QuBE, 16, 17, 22, 41
qxbf, 18

resolution, 7
restart, 21

SAT solver, 9, 25
search space splitting, 11
search-based QBF solving, 3, 8–11, 30
single quantification level scheduling, 28
Skolem function, 31
solution analysis, 22
solution learning, 8
SQLS, 23, 25
SqueezBF, 22, 23, 29
strongly connected component, 17

trivial falsity, 25
trivial truth, 25

universal reduction, 7, 9, 11

variable dependency, 4
variable-activity scaling, 22
VSIDS, 27

young brothers wait scheduling, 26