HordeQBF: A Modular and Massively Parallel QBF Solver

Tomáš Balyo[1][⋆] and Florian Lonsing[2][⋆⋆]

[1] Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
[2] Knowledge-Based Systems Group, Vienna University of Technology
Vienna, Austria

Abstract. The recently developed massively parallel satisfiability (SAT) solver HordeSAT was designed in a modular way to allow the integration of any sequential CDCL-based SAT solver in its core. We integrated the QCDCL-based quantified Boolean formula (QBF) solver DepQBF in HordeSAT to obtain a massively parallel QBF solver—HordeQBF. In this paper we describe the details of this integration and report on results of the experimental evaluation of HordeQBF's performance. HordeQBF achieves superlinear average and median speedup on the hard instances of the 2014 QBF Gallery (Competition).

1 Introduction

HordeSAT [3] is a modular massively parallel SAT solver which allows the integration of any sequential CDCL-based SAT solver in its core. This enables the transfer of advancements in CDCL SAT solving to a parallel setting. Experiments showed that HordeSAT can achieve superlinear average speedup on hard benchmarks.

The logic of quantified Boolean formulas (QBFs) extends SAT by explicit quantification of propositional variables. Problems in complexity classes beyond NP, particularly PSPACE-complete problems in domains like, e.g., formal verification, reactive synthesis, or planning, can naturally be encoded as QBFs.

QBF solvers based on QCDCL, the QBF-specific variant of CDCL, apply techniques similar to CDCL SAT solvers. Thanks to this fact, it is possible to replace the SAT solver in the core of HordeSAT by *any* QCDCL QBF solver. Thereby, it is not necessary to change the framework of HordeSAT which controls the sharing of learned information and the execution of the core solver instances.

We integrated the latest public version 5.0 of the QCDCL-based solver DepQBF [18] in HordeSAT to obtain the massively parallel QBF solver HordeQBF. We present the implementation of HordeQBF, which is not tailored towards the use of DepQBF as a core solver, and evaluate its scalability on a computer cluster with 1024 processor cores. Experiments using the application benchmarks of the 2014 QBF Gallery show that HordeQBF achieves superlinear average and median speedup for hard instances.

2 Preliminaries

We consider closed QBFs $\psi := \Pi.\phi$ in *prenex CNF (PCNF)* consisting of a quantifier-free CNF ϕ over variables V and a *quantifier prefix* $\Pi := Q_1 V_1 \ldots Q_n V_n$. QBF solving with clause and cube learning (*QCDCL*) [9,15,27], also called constraint learning, is a generalization of *conflict-driven clause learning (CDCL)* for SAT. The variables in a PCNF ψ are assigned by *decision making*, *unit propagation*, and *pure literal detection*. Assignments by decision making have to follow the prefix ordering from left to right. If a clause if falsified under the current assignment A, then a *learned clause* C is derived from ψ by *Q-resolution* [14] and added conjunctively to ψ. If all clauses are satisfied under A, then a *learned cube* is constructed from A and added disjunctively to ψ. Learned cubes may also be derived by *term resolution* [9], a variant of Q-resolution applied to previously learned cubes. After a new clause or cube has been learned, assignments are retracted during *backtracking*. QCDCL terminates if and only if the *empty clause (cube)* is derived during learning, indicating that ψ is unsatisfiable (satisfiable).

3 Related Work

Approaches to parallel QBF solving are based on shared and distributed memory architectures. PQSolve [7] is an early parallel DPLL [5] solver without knowledge sharing. It comes with a dynamic master/slave framework implemented using the message passing interface (MPI). Search space is partitioned among master and slaves by variable assignments. QMiraXT [16] is a multithreaded QCDCL solver with search space partitioning. PAQuBE [17] is an MPI-based parallel variant of the QCDCL solver QuBE [8]. Clause and cube sharing in PAQuBE can be adapted dynamically at run time. Search space is partitioned like in the SAT solver PSATO based on *guiding paths* [25]. The MPI-based solver MPIDepQBF [13] implements a master/worker architecture without knowledge sharing. A worker consists of an instance of the QCDCL solver DepQBF [19]. The master balances the workload by generating subproblems defined by variable assignments (assumptions), which are solved by the workers. Parallel solving approaches have also been presented for quantified CSPs [24] and non-PCNF QBFs [21].

HordeQBF is a parallel portfolio solver with clause and cube sharing. Whereas sequential portfolio solvers like AQME [23] include different QBF solvers, Horde-QBF integrates instances of the same QCDCL solver (i.e., DepQBF). Unlike MPIDepQBF, HordeQBF does not rely on search space partitioning. Instead, the parallel instances of DepQBF are diversified by different parameter settings.

4 The HordeSAT Parallelization Framework

HordeSat is a portfolio SAT solver with clause sharing [3]. It can be viewed as a multithreaded program running several instances of a sequential SAT solver and communicating via messages [3] with other instances of the same program.

---

[3] The communication is implemented using the Message Passing Interface (MPI) [10].

The parallelization framework has three main tasks: to ensure that the core solvers are diversified, to handle the clause exchange, and to stop all the solvers when one of them has solved the problem. To communicate with the core solvers it uses an API, which is described in detail in the HordeSAT paper [3]. Since the HordeQBF interface is identical, we only briefly list the most relevant methods:
void diversify(int rank, int size): This method tells the core solver to diversify its settings. The specifics of diversification are left to the solver. The description for DepQBF is given in the following section.
void addLearnedClause(vector<int> clause): This method is used to import learned clauses (and cubes) received from other solvers of the portfolio.
void setLearnedClauseCallback(LCCallback* callback): This method sets a callback class that will process the clauses (and cubes) shared by this solver.

5   QBF Solver Integration

In parallel QCDCL-based QBF solving, learned cubes may be shared among the solver instances in addition to learned clauses. Although HordeSAT does not provide API functions dedicated to cube sharing, its available API readily supports it. We describe the integration of the QCDCL-based QBF solver DepQBF[4] in HordeQBF, which applies to any QCDCL-based QBF solver.

We rely on version 5.0 of DepQBF which comes with a dynamic variant of *blocked clause elimination (QBCE)* [18] for advanced cube learning. QBCE allows to eliminate redundant clauses from a PCNF [11]. Dynamic QBCE is applied frequently during the solving process. If all clauses in the PCNF are satisfied under the current assignment or removed by QBCE, then a cube is learned.

DepQBF features a sophisticated analysis of variable dependencies in a PCNF [19,20] to relax the linear ordering of variables in the prefix. For HordeQBF, however, we disabled dependency analysis to avoid its computational cost.

We modified DepQBF as follows to integrate it in HordeQBF. Learned constraints are exported to the master process right after they have been learned. The master does not distinguish between learned clauses and cubes but treats them as sorted lists of literals. We add special marker literals to learned clauses and cubes to distinguish between them at the time when the master provides the workers with sets of shared learned constraints.

In DepQBF we check whether shared constraints are available for import after a restart has been carried out. To this end, we modified the restart policy of DepQBF to always backtrack to decision level zero. This is different from the original restart policy of DepQBF [19], where the solver backtracks to higher decision levels depending on the current assignment. After a restart, available shared constraints are imported, the watched data structures are updated, and QCDCL continues by propagating unit literals resulting from imported constraints.

Every instance of DepQBF receives a random seed from the master and diversifies the solving process as follows. The values of variables in the *assignment*

Core	Parallel	Both	Speedup All			Speedup Big		
Solvers	Solved	Solved	Avg.	Tot.	Med.	Avg.	Tot.	Med.
2×4×4	513	483	622	107.30	0.82	3328	127.36	303.26
4×4×4	516	484	667	137.36	0.92	3893	176.27	458.34
8×4×4	523	492	748	128.35	0.96	4655	175.26	553.53
16×4×4	527	493	754	140.37	0.96	5154	236.18	1449.28
32×4×4	531	496	780	132.41	0.96	6282	269.87	2461.84
64×4×4	532	496	762	141.99	0.89	6702	307.29	2557.54

Table 1. The speedup of HordeQBF configurations relative to DepQBF. The second column is the number of instances solved by HordeQBF, the third is the number of instances solved by both DepQBF and the HordeQBF. The following six columns contain the average, total, and median speedups for either all the instances solved by HordeQBF or only big instances (solved after $10 \times \#$cores seconds by DepQBF).

cache [22] are initialized at random. In general, decision variables are assigned the cached value (if any). The assignment cache is updated with values assigned by unit propagation and pure literal detection. As an effect of random initialization, the first value assigned to a decision variable is always a random value. Parameters of *variable activity scaling* are set at random. DepQBF implements variable activities similar to MiniSAT [6]. Additionally, the amount (percentage) of learned constraints that are removed periodically is initialized at random. DepQBF stores learned clauses and cubes in separate lists with certain capacities. If a list has been filled during learning then less frequently used constraints are removed and the capacity of the list is increased. DepQBF implements a *nested restart scheme* similar to PicoSAT [4], the parameters of which are randomly selected. Variants of *dynamic QBCE* [18] are enabled at random, including switching off dynamic QBCE at all, applying QBCE only as a preprocessing, or inprocessing step. Finally, applications of *long-distance resolution* [1,26], an extension of traditional Q-resolution [14] used to derive learned constraints, are toggled at random.

6   Experimental Evaluation

To examine our portfolio-based parallel QBF solver HordeQBF we performed experiments using all the 735 benchmark problems from the application track of the 2014 QBF Gallery (Competition) [12]. We compared HordeQBF with DepQBF, which is the QBF solver in the core of HordeQBF.

The experiments were run on a cluster with nodes having two octa-core 2.6 GHz Intel Xeon E5-2670 processors (Sandy Bridge) and 64 GB of main memory. Each node has 16 cores and we used 64 nodes which amounts in the total of 1024 cores. The nodes communicate using an InfiniBand 4X QDR Interconnect and use the SUSE Linux Enterprise Server 11 (x86_64) (patch level 3) operating system. HordeQBF was compiled using the icpc compiler version 15.0.2.
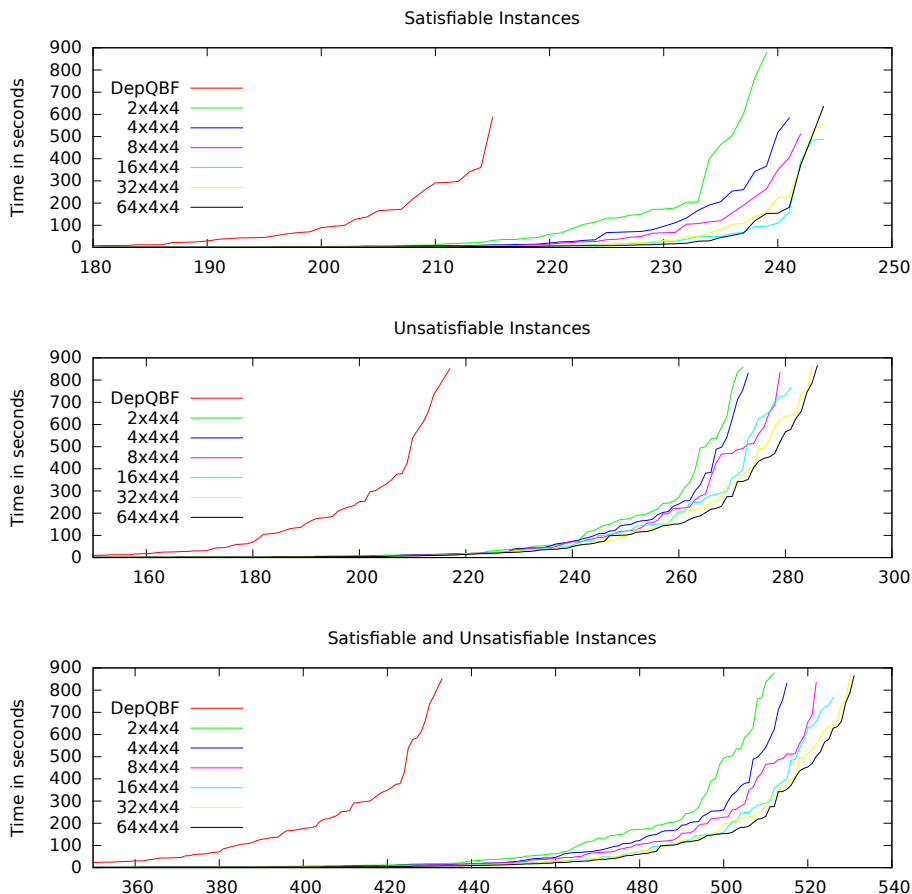
Fig. 1. Cactus plots for the benchmarks solved under 900 seconds by DepQBF and various configurations of HordeQBF. The left regions of the plots (containing easy instances) are omitted.

The complete source code and detailed experimental results are available at `http://baldur.iti.kit.edu/hordesat/`.

We ran experiments using 2, 4, ..., 64 cluster nodes. On each node we ran four processes with four threads each, which amounts to 16 core solver (DepQBF) instances per node. The results are summarized in Figure 1 using cactus plots. We can observe that increasing the number of cores is beneficial for both SAT and UNSAT instances since the number of solved instances steadily increases and runtimes are reduced.

However, it is not easy to see from a cactus plot whether the additional performance is a reasonable return on the invested hardware resources. Therefore we include Table 1 in order to quantify the overall scalability of HordeQBF. We
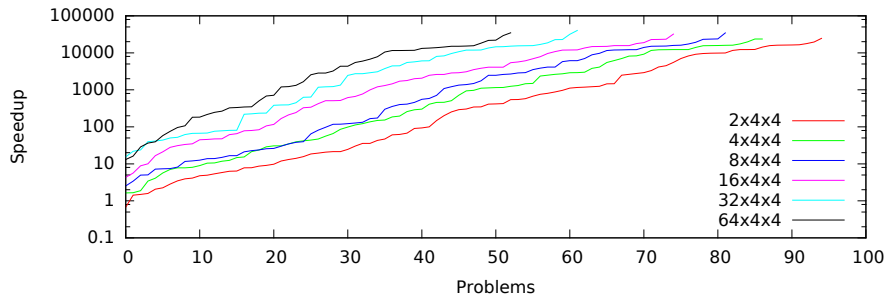
5

Fig. 2. Distribution of speedups on the "big instances" (solved after $10 \times \#$cores seconds by DepQBF – the data corresponding to Columns 7–9 of Table 1).

compute speedups for all the instances solved by the parallel solver. We ran DepQBF with a time limit $T = 50\,000s$ and for the instances it did not solve we generously assume that it would solve them if given $T + \epsilon$ seconds and use the runtime of T for speedup calculation. The parallel configurations have a time limit of $900s$. Columns 4, 5, and 6 of Table 1 show the average, total (sum of sequential runtimes divided by the sum of parallel runtimes) and median speedup values respectively. While the average and total speedup values are high, the median speedup is below one.

Nevertheless, these figures treat HordeQBF unfairly since the majority of the benchmarks is easy (solvable under a minute by DepQBF) and it makes no sense to use large computer clusters to solve them. In parallel computing, it is usual to analyze the performance on many processors using *weak scaling* where one increases the amount of work involved in the considered instances proportionally to the number of processors. Therefore in columns 7–9 we restrict ourselves to "big instances" – where DepQBF needs at least $10 \times$(the number of cores used by HordeQBF) seconds to solve them. The average, total and median speedup values get significantly larger and in fact we obtain highly superlinear average and median speedups. Figure 2 shows the distribution of speedups for these instances, it also reveals how many instances (x-axis) qualify as "big instances".

7 Conclusion

We showed that QBF solving can be successfully parallelized using the same techniques as for massively parallel SAT solving. Our parallel QBF solver HordeQBF achieved superlinear total and median speedups for hard instances, i.e., instances where parallelization makes sense.

As future work it would be interesting to consider further variants of Q-resolution systems [2] (apart from traditional [14] and long-distance resolution [1,26]) as a means of diversification in HordeQBF, which would amount to a combination of QBF proof systems with different power. Further, it may be promising to equip HordeQBF with search space partitioning as in MPIDepQBF [13].

References

1. Balabanov, V., Jiang, J.R.: Unified QBF certification and its applications. Formal Methods in System Design 41(1), 45–65 (2012)
2. Balabanov, V., Widl, M., Jiang, J.R.: QBF Resolution Systems and Their Proof Complexities. In: SAT. LNCS, vol. 8561, pp. 154–169. Springer (2014)
3. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A Massively Parallel Portfolio SAT Solver. In: SAT. LNCS, vol. 9340, pp. 156–172. Springer (2015)
4. Biere, A.: PicoSAT Essentials. JSAT 4(2-4), 75–97 (2008)
5. Cadoli, M., Giovanardi, A., Schaerf, M.: An Algorithm to Evaluate Quantified Boolean Formulae. In: AAAI. pp. 262–267. AAAI Press / The MIT Press (1998)
6. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003)
7. Feldmann, R., Monien, B., Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In: Proc. of the 17th Nat. Conference on Artificial Intelligence (AAAI 2000). pp. 285–290. AAAI (2000)
8. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE7.0. JSAT 7(2-3), 83–88 (2010)
9. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. JAIR 26, 371–416 (2006)
10. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing 22(6), 789–828 (1996)
11. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause Elimination for SAT and QSAT. J. Artif. Intell. Res. (JAIR) 53, 127–168 (2015)
12. Janota, M., Jordan, C., Klieber, W., Lonsing, F., Seidl, M., Van Gelder, A.: The QBFGallery 2014: The QBF Competition at the FLoC Olympic Games. Journal on Satisfiability, Boolean Modeling and Computation 9, 187–206 (2016)
13. Jordan, C., Kaiser, L., Lonsing, F., Seidl, M.: MPIDepQBF: Towards Parallel QBF Solving without Knowledge Sharing. In: SAT. LNCS, vol. 8561, pp. 430–437. Springer (2014)
14. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Inf. Comput. 117(1), 12–18 (1995)
15. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: TABLEAUX. LNCS, vol. 2381, pp. 160–175. Springer (2002)
16. Lewis, M., Schubert, T., Becker, B.: QMiraXT – a multithreaded QBF solver. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV) (2009)
17. Lewis, M., Schubert, T., Becker, B., Marin, P., Narizzano, M., Giunchiglia, E.: Parallel QBF Solving with Advanced Knowledge Sharing. Fundamenta Informaticae 107(2-3), 139–166 (2011)
18. Lonsing, F., Bacchus, F., Biere, A., Egly, U., Seidl, M.: Enhancing Search-Based QBF Solving by Dynamic Blocked Clause Elimination. In: LPAR. LNCS, vol. 9450, pp. 418–433. Springer (2015)
19. Lonsing, F., Biere, A.: DepQBF: A Dependency-Aware QBF Solver. JSAT 7(2-3), 71–76 (2010)
20. Lonsing, F., Biere, A.: Integrating Dependency Schemes in Search-Based QBF Solvers. In: SAT. LNCS, vol. 6175, pp. 158–171. Springer (2010)
21. Mota, B.D., Nicolas, P., Stéphan, I.: A new parallel architecture for QBF tools. In: Proc. of the Int. Conf. on High Performance Computing and Simulation (HPCS 2010). pp. 324–330. IEEE (2010)

22. Pipatsrisawat, K., Darwiche, A.: A Lightweight Component Caching Scheme for Satisfiability Solvers. In: SAT. LNCS, vol. 4501, pp. 294–299. Springer (2007)
23. Pulina, L., Tacchella, A.: Aqme'10. JSAT 7(2-3), 65–70 (2010), `http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_5_Pulina.pdf`
24. Vautard, J., Lallouet, A., Hamadi, Y.: A Parallel Solving Algorithm for Quantified Constraints Problems. In: ICTAI. pp. 271–274. IEEE Computer Society (2010)
25. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. J. Symb. Comput. 21(4), 543–560 (1996)
26. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean Satisfiability solver. In: ICCAD. pp. 442–449. ACM / IEEE Computer Society (2002)
27. Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: CP. LNCS, vol. 2470, pp. 200–215. Springer (2002)