Evaluation of Knowledge Sharing Strategies in a Parallel QBF Solver*

Paolo Marin[1], Matthew Lewis[2], Tobias Schubert[2], Massimo Narizzano[1], Bernd Becker[2], and Enrico Giunchiglia[1]

[1] University of Genova
Genova, Italy
name.surname@unige.it

[2] University of Freiburg
Freiburg, Germany
surname@informatik.uni-freiburg.de

Abstract

In this paper we examine the effect that different knowledge sharing strategies have on the performance of our parallel QBF Solver PaQuBE. This new Master/Slave MPI based solver leverages the additional computational power that can be exploited from modern computer and system architectures, to solve more relevant instances and faster than previous generation solvers. Knowledge sharing plays a critical role in the performance of PaQuBE. However, due to the overhead associated with sending and receiving MPI messages, and the restricted communication/network bandwidth available between solvers, it is essential that we optimize not only which information is shared, but how it is shared. In this context, we compare multiple conflict clause and solution cube sharing strategies, and finally show that an adaptive method works best. Additionally, compression of solution cubes was explored which reduced the system time associated with message passing while also reducing network traffic.

1 Introduction

The current generation of Boolean Satisfiability (SAT) and Quantified Boolean Formula (QBF) solvers have become quite powerful. Both are now able to solve many practically relevant problems. QBF however, allows researchers to more naturally and compactly encode a wider range of problems that for instance are encountered in Black Box or Partial Circuit Verification [15], Bounded Model Checking [5], and AI planning [19], than in SAT. Since QBF problems are generally more difficult (PSPACE-Complete

*This work appeared as *"Comparison of Knowledge Sharing Strategies in a Parallel QBF Solver"* in *Proceedings of The 2009 High Performance Computing & Simulation (HPCS 2009) Conference.*

vs. NP-Complete), they require dedicated algorithms and increased computation power to solve relevant instances. In this context, using multiprocessor systems and parallel algorithms is an interesting solution.

In the domain of parallel solvers, research focuses on subproblem generation, and knowledge sharing. Here we are concentrating on the latter. On parallel QBF (or SAT) solvers, using static criteria for selecting what information should be shared results in many messages being sent, but actually very little good information being shared. This paper highlights this problem, and tries to provide some new ideas to improve knowledge sharing within an Message Passing Interface (MPI) [20] based system.

The next section will describe the QBF problem, and how sequential and parallel QBF solvers work. Section 3 will talk about the design and implementation of *PaQuBE*. The knowledge sharing strategies that were tested are discussed in Section 4 . Section 5 will cover the performance results. Finally, Section 6 will conclude this paper.

2 QBF and DPLL Solver Overview

There are many ways to encode a QBF problem, but in our context, they are defined in Conjunctive Normal Form (CNF). A problem in CNF form starts with a variable definition. The variable definition quantifies each variable (either existentially or universally), and assigns each variable to a specific quantification level. Once the variable definition is complete, a set of clauses is given that defines the problem. More formally, a *QBF* is an expression of the form:

$$\varphi = Q_1 z_1 Q_2 z_2 \dots Q_n z_n \Phi \qquad (n \geq 0) \tag{1}$$

Here, every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$, $z_1, \dots, z_n$ are distinct sets of variables, and Φ is a propositional formula. $Q_1 z_1 \dots Q_n z_n$ is defined as the *prefix*, and Φ, the propositional formula, would contain a set P of clauses. While a *variable* is defined as an element of P, an occurrence of that variable or its negation in a clause is referred to as a *literal*. In the following, the literal $\bar{l}$ is defined as the negative occurrence of the varable $|l|$ in P, and l is the positive occurrence. In the following, we also use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction, respectively. For example, an entire problem definition might be as follows:

$$\exists x_1 \forall y \exists x_2 \{\{\overline{x}_1 \vee \overline{y} \vee x_2\} \wedge \{x_1 \vee \overline{y}\} \wedge \dots\} \tag{2}$$

We say that (1) is in *Conjunctive Normal Form* (CNF) when Φ is a conjunction of *clauses*, where each clause is a disjunction of literals as shown in (2). And that (1) is in *Disjunctive Normal Form* (DNF) when Φ is a disjunction of *cubes*, where each cube is a conjunction of literals (also known as Solution Term) . We use *constraints* when we refer to clauses and cubes indistinctly. We also define (i) the *level of a variable* z_i, to be 1 + the number of alternations $Q_j z_j Q_{j+1} z_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$; the (ii) *level of a literal* l, to be the level of $|l|$; and the (iii) *level of the formula*, to be the level of z_1.

So, for example, in equation (2), x_2 is existential and is quantified on level 1, y is universal and is on level 2, x_1 is existential and is on level 3.

2.1 Sequential QBF Solver

There are many types of sequential QBF solvers. Most solvers like QMiraXT [18], QuBE [12], yQuaffle [21], are in principal based on the DPLL algorithm [4]. Others, like Quantor [2], try to resolve and expand the formula until no universally quantified variables remain. This allows them to send their remaining existentially quantified problems to a SAT solver. On the other hand, solvers like sKizzo [1] do the opposite of Quantor, and use symbolic skolemization to eliminate the existentially quantified variables.

Now, while there are many ways to solve a QBF problem, this paper focuses on *PaQuBE* which is based on the DPLL algorithm [4]. A DPLL solver would start by preprocessing the input formula (CNF), and then, using a heuristic, a variable would be selected and assigned a value (TRUE or FALSE). In QBF, the decision heuristic is restricted to choosing variables on the first quantification level. Only when all the variables on this level are defined, can the heuristic move on to the next level. Once a decision is made, a Boolean Constraint Propagation (BCP) procedure is run to find consequences or implications of that decision. If the BCP procedure completes and no conflicts are found, the decision procedure is run again.

However, if a conflict is found, a conflict analysis procedure is run in order to find the reason for the conflict. This procedure will try to resolve the conflict by backtracking to a previous decision level. It will then learn a conflict clause that allows the solver to avoid this part of the search space in the future. However, if the conflict cannot be resolved, the problem is unsatisfiable. On the other hand, whenever a solution is found, an initial reason can be computed in order to run the above conflict analysis procedure almost symmetrically, thus recording a solution cube. If the solution cannot be resolved, the problem is satisfiable.

For this paper, the more interesting advancements relate to this. Conflict/Solution analysis with non-chronological backtracking [7, 12, 16, 23, 11] allows a SAT or QBF solver to produce new clauses and/or cubes that can significantly truncate the search space. In a parallel solver, this information can be shared between each of the sequential solvers.

2.2 Parallel QBF Solver

In our context, a parallel QBF solver consists of multiple copies of a sequential solver. Each sequential solver (in the total parallel solver) works in the same manner as described in Section 2.1. However, instead of working on the entire problem, each individual solver is given a small part of the original problem. This can be accomplished by selecting a decision variable and telling each solver to search opposite assignments of that variable. This method is referred to as the Guiding Path method in SAT and it was first introduced by PSATO [22]. However, when using this method to produce subproblems in QBF, a more elaborate mechanism must be in place to keep track of which parts of the search space are currently being searched, and which parts have already been proved satisfiable or unsatisfiable. For more information on this refer to [18].

2.3 Previous Parallel QBF Solver Work

The parallelization of SAT has been well studied, e.g. in [22, 3]. However, there is only one Message Passing based QBF solver that we are aware of [6] (PQSOLVE). PQSOLVE was based on the basic DPLL algorithm, without conflict or solution analysis as these techniques did not exist when it was published. Recently, the threaded parallel SAT solver MiraXT was modified so that it could directly handle QBF formulas [18]. Its tight integration of threads allows significantly more knowledge sharing than an MPI design. Because of QMiraXT's shared memory architecture, each solver thread can select the clauses it wishes to take, based on its current status. This is the optimal solution, but a thread based design has scalability issues. With larger MPI solvers no shared memory is available, so each solver must select which clauses or terms it thinks are the best to share. In most implementations, each individual solver is unaware of the status of the other solvers. For example, in the parallel SAT solver GridSAT [3], a predefined maximum clause length of 3 was used as the static criteria for sharing clauses. Even newer solvers such as PMSat [8] still do this. In [14] the authors propose a dynamic approach, but this pertains only to SAT. Furthermore, solution cube sharing has not yet been explored. The first parallel QBF solver QMiraXT only performed solution analysis, but recorded and shared no solution cubes.

3 PaQuBE Design Overview

We now present a quick overview of our distributed QBF proving algorithm *PaQuBE*. *PaQuBE* is based on *QuBE* which is a search based QBF Solver that uses lazy data structures for both unit clauses propagation and for pure literals detection [12]. It also features conflict and solution non-chronological backtracking and learning[1]. This overview is given in order to present the framework in which *PaQuBE* uses for message passing and knowledge sharing. It is necessary to understand the discussion of knowledge sharing in Section 4. For a more detailed description of *PaQuBE* refer to [17].

3.1 General Properties

PaQuBE was implemented using a Master/Slave Model, where one process is dedicated to be the master, and $n-1$ are acting as slaves that actually perform the solving. n here represents the total number of processes running on the system. An illustration, using three clients, is given in Figure 1. In our implementation, the role of the master is to control requests for new subproblems, ensuring that the entire search space is searched. *PaQuBE* uses the the Single Quantification Level Scheduling (SQLS) algorithm from [18] to do this.

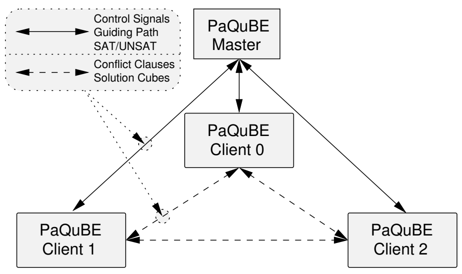

Figure 1: PaQuBE Design

In *PaQuBE*, the master sleeps most of the time, and when working there is at least one inactive slave. This allows it to be run alongside other processes without really needing a dedicated CPU. Also, in contrast to many other parallel MPI based SAT solvers, the knowledge sharing mechanism does not involve the master. Instead, the clients are total responsible for sharing knowledge as is shown by Figure 1. Finally, the entire communication has been realized using MPICH2 [13], an implementation of the Message Passing Interface standard [20].

3.2 Implementation of Knowledge Sharing

As stated above, *PaQuBE* slaves can share both learnt clauses and cubes. As conflict learning made SAT/QBF Solvers able to solve real world problems, solution learning in QBF can help as well [11]. Moreover, it is well known that computing initial reasons for backjumping from a solution (terms or cubes) is far more expensive than the conflict case (see [23] and [9] for more detailed

[1]*QuBE*6.5 is a composition of the preprocessor *sQueezeBF* and the core solver. Our work focuses on the core solver, but the formula output by the preprocessor has been taken into account.

considerations). As a consequence, sharing small and already computed solution terms may speed up the search. With future considerations in mind, and in order to save part of the time (latency and transmission time) needed to send these large messages in general, clauses are packed into bundles of a fixed size, and terms are packed and compressed, with the aim of filling without exceeding the capacity of a TCP packet. This bundles are prepared and broadcasted after a fixed number of propagated literals. Furthermore, to reduce network traffic, *PaQuBE* uses the design shown in Figure 1 in which the exchange of conflict clauses does not involve the master process.

4 Knowledge Sharing Strategies

We implemented and tested experimentally different strategies for sharing clauses and cubes between slaves. Some strategies use static criteria such as size, while others are adaptive. In the case of cubes, further criteria were tested that are based on quantifier alternations. However, in order to provide a light efficient implementation of these criteria, a bucket sort system was used. Instead of searching the clause/cube database for the best choices when we want to send a message, we add the clauses/cubes to the different buckets when they are produced, depending on the knowledge sharing criteria used.

For example, *PaQuBE* uses 3 buckets for conflict clauses and 3 buckets for solution cubes. If we use clause length as our criteria, a new clause will be placed in a particular bucket based on its length. In our implementation the first bucket contains all clauses with a length < 5, the second is for clauses with a length of $5 - 10$, and a third for larger clauses that still meet the criteria discussed below.

When the time comes to share this information, the clauses are taken from the first bucket, then the second bucket, and finally the third bucket until the 20 clause packet is full. This system allows us to not sort and search the list of new clauses which can be quite large. Finally, once shared, the buckets are emptied and the process starts again.

4.1 Selecting Information to Take

Now, when receiving clauses or cubes, slaves only add the ones that really aid their search. These include clauses that are conflicting or directly produce implications. Furthermore, really short clauses are also added as they may still be useful in the near future or on new subproblems, and they are easier to process by the BCP procedure.

This eliminates adding many useless clauses or terms, while providing a balance between the knowledge sharing and the number of clauses the BCP procedure must evaluate. This is the main limitation of an MPI based parallel solver. When a slave selects constraints to be shared, it is not aware of their usefulness to other slaves. This is because slaves are not aware of other slaves' current status or subproblems. In order to exchange this information and keep it up-to-date would imply either too many messages or too great latency if updated just before sharing. Selecting the constraints in this way however, would allow us to share even larger ones more effectively, even if less knowledge in total was shared. There are some interesting tradeoffs here that we are currently developing.

4.2 Conflict Clause Sharing Strategies

Conflict clauses are shared depending on their length. The threshold is calculated as a percentage of the number of variables occurring in the formula. The three strategies *CS1*, *CS2*, and *CS3* compute a fixed upper bound, while *CSADP* uses an initial upper bound (equal to *CS2*) that can be dynamically turned up or down depending on the amount of clauses that have been produced. The four different strategies are as follows:

1. *CS1* - Shared if clause length $\leq$ 6%, taken if clause length $\leq$ 2% or conflicting or unit.
2. *CS2* - Shared if clause length $\leq$ 11%, taken if length $\leq$ 3% or conflicting or unit.
3. *CS3* - Shared if clause length $\leq$ 15%, taken if length $\leq$ 5% or conflicting or unit.
4. *CSADP* - Shared with an adaptive clause length from 0 to 40%, taken if length $\leq$ 3% or conflicting or unit. The initial threshold is set to be the same of *CS2*.

For all these strategies, the bucket sort discussed above is used. This means that really short clauses still have a high priority and will always be chosen over longer clauses. Only when the third bucket is needed, which is often the case as small clauses are quite rare and we try to send packets of 20 clauses at time, do the criteria for *CSx* and *CSADP* play a large role.

As for the adaptive clause length criteria (*CSADP*), this starts using the same threshold values as *CS2*. When we start to share clauses, the buckets are checked as discussed before. If the $3rd$ bucket does not contain enough clauses that meet the current criteria to fill the packet (20 clauses), the threshold is increased by 10%. On the other hand, if the third bucket contains more than enough clauses to fill a packet by itself ($\geq$ 20), the threshold is decreased by 10%. In all the other cases, the threshold does not change.

4.3 Solution Cube Sharing Strategies

Solution cubes are shared following 4 distinctly different strategies. The first group, *SS1* and *SS2*, work in the same manner as *CS1* and *CS2* but using different (larger) values. Also, in *QuBE*, cubes are minimized by removing all variables quantified by the innermost quantifier. This means the selection strategies for cubes are based on the number of variables quantified from the 2^{nd} to the highest quantification level. The next strategies for cubes are labeled *SSQA* and *SSFU*. They compute a stricter cube length bound with respect to *SS1* or *SS2*, but they allow for larger cubes (up to a limit calculated in a similar fashion for *SSx*) to be sent if the cubes contain only universally quantified variables from the first universal quantification levels (*SSQA*), or contain no more than a fixed number of universally quantified variables (*SSFU*). Finally, *SSADP* is similar to *CSADP*, but for cubes.

1. *SS1* - Shared if cube length $\leq 15\%$, taken if length $\leq 6\%$ or conflicting or unit.
2. *SS2* - Shared if cube length $\leq 18\%$, taken if length $\leq 8\%$ or conflicting or unit.
3. *SSQA* - Shared if cube length $\leq 9\%$, taken if length $\leq 8\%$ or conflicting or unit. Also shared if length is $\leq 35\%$ and its literals are bound at the first two universal quantification levels.
4. *SSFU* - Shared if cube length $\leq 9\%$, taken if length $\leq 8\%$ or conflicting or unit. Also shared if length is $\leq 35\%$ and it contains no more than 2 universal variables.
5. *SSADP* - Shared with adaptive cube length from 0 to $\leq 50\%$, taken if length $\leq 8\%$ or conflicting or unit. The initial threshold is set to be the same of *SS2*.

For all these strategies, the bucket system was used again, however, the length of the cubes for the first two buckets was doubled. Instead of using a clause length of 5 and 10 for each bucket, a cube length of 10 and 20 were used for the first two buckets. The third bucket again contains the remaining cubes that meet the maximum criteria allowed.

For *SSQA* and *SSFU*, the selection mechanism is a bit more complicated. Long cubes that meet these criteria are put into the second bucket (instead of third) to give them a higher priority. This means that while really short cubes still have the highest priority, long cubes with few universally quantified variables or that only contain universally quantified variables bound to the first two levels, are shared over shorter but more general cubes. Lastly, *SSADP* works in the same manner as *CSADP*, with the exact same rules, but with respect to the buckets used for cube sharing.

4.4 Solution Cube Compression

The algorithm used for compressing terms works on the assumption that these terms share many literals, in particular those quantified at the highest levels. Therefore, if the literals occurring in these terms are sorted according to the prefix order, in every block of terms we can effectively detect and avoid sending the common part of each. Moreover, each literal is encoded using only two bits. This encoding allows us to communicate that a literal (*i*) occurs with a positive polarity (01), (*ii*) occurs with negative polarity (11), or (*iii*) does not occur in this term (00). The remaining allowable value is used as a marker for the end of the term. Finally, after converting all the selected terms, we put the complete first term into the packet. Then, for the following terms, we only include the term's differing tail, and an offset pointing to where this term starts to differ from the first one. For example, consider the following formula: $\varphi = \forall y_1 y_2 \exists x_1 x_2 \forall y_3 \exists x_3 \varphi$. Excluding the innermost existential variables (those bounded at the lowest quantification level) 5 atoms may occur in a term (because of minimization). Now, let's say a solver learns: $\{y_1, \neg y_2, x_1, y_3\}$, $\{y_1, \neg y_2, x_1, \neg x_2, \neg y_3\}$. Their 2−bit encodings are, respectively: {`01`|`11`|`01`|*00*|*01*}, {`01`|`11`|`01`|*11*|*11*}. Only the last 2 literals (highlighted in *italic*) differ. We say: *"the difference begins at the* 4*th* *position"*. Then, the sent message will be: {`01`|`11`|`01`|`00`|`01`|**4**|`11`|`11`}. Finally, the comparison here between terms has been done literal by literal (pairs of bits), but for the sake of efficiency in *PaQuBE* this is done between sets of 16 literals (32 bit blocks).

5 Experimental Results

To evaluate the performance of *PaQuBE* and the effectiveness of our ideas, we ran multiple experiments on a selected pool of fixed-structure instances from *qbflib* [10]. All instances from *qbflib* for which *QuBE* (the sequential solver) required between 10 and 600 seconds were used, plus some incrementally harder instances. In total, 283 benchmarks problems, from over 20 families, were used.

The benchmarking machine used in this section contains two Dualcore AMD Opteron 280 processors. Each processor runs at 2.4 GHz, and is connected to 2 GB of local memory (4 GB in total). This machine runs a 64 bit version of the Linux 2.6.24 kernel, and supports the MPICH 2-1.0.8 library. This AMD system provides significantly more performance for message passing than other distributed systems. On a larger cluster, *PaQuBE*'s knowledge sharing would have to be scaled down accordingly with the available bandwidth. Finally, it should be noted that *PaQuBE*'s average CPU utilization when run with all 4 processors was over 97% on these benchmarks. This means that all solvers were normally active solving the problem and sharing information.

In Table 1 we first compare the different knowledge sharing strategies as described in Section 4. Because the exact solve time of a parallel solver is non-deterministic, each of the parallel solvers were run twice, and the averages of those two runs are shown. The columns labeled $\#CCS/s$ and $\#CCT/s$ are for conflict clauses shared per second, and clauses taken per second per client. $\#SCS/s$ and $\#SCT/s$ are the same for solution cubes. If knowledge sharing was perfect, the clauses/cubes shared would equal the number taken. In brackets, the average size of the clause/cube that were shared and taken is also shown. $Time$ is the wall clock time used by the solver, and $\#PS$ is the number of problems solved by either run of the solver.

First, we show the performance difference between the sequential solver *QuBE* and *PaQuBE 4P-NOS* which is our parallel solver with sharing disabled. Here we still get good speedup (with respect to the number of problems solved, $\#PS$, and $Time$) without knowledge sharing. Next, with conflict clause sharing enabled, the three static strategies *CS1-3* provide roughly no added benefit. Only when the adaptive method is used, does conflict clause sharing actually help. The same is true with solution cube sharing. This is interesting as most current state-of-the-art solvers use static criteria for sharing clauses. As for *SSQA* and *SSFU*, which depend on quantifier alternations, these types of strategies do not seem to help, although from a logical standpoint they seem resonable.

The fully adaptive cube and clause sharing strategy provides the best performance. This is shown in *PaQuBE 4P-CSSSADP*. The *4P-CSSSADP* version does not only reduce the total solving time by 2000s, but also solves 7 more instances compared to *PaQuBE 4P-NOS*. Furthermore, *4P-CSSSADP* performs better than the optimized static approach originally used in *PaQuBE* (PaQuBE 4P-ver-[17]).

As for a general pattern, it is interesting to note that in all cases when sharing conflict clauses (and to a similar extent solution cubes), the amount shared is significantly more than the amount taken. This means much of the information shared by each solver does not really help the other solvers. This is shown in more detail in Table 2.

In Table 2 we now show the results of our best solver *PaQuBE 4P-CSSSADP* on each family. It can be seen that different families of benchmarks perform better than others in the parallel sense, providing variable speedup (4P×). With respect to benchmarks like `katz`, in which we achieve super linear speedup, this is basically attributed to the fact that one of the 4 clients received a subproblem that produced a conflict that showed that the entire problem was unsatisfiable. This again is an advantage of a parallel solver. Decision heuristics are not perfect, and by adding more clients, we have a better chance of sending the solver to a more fruitful part of the search space. One reason for the poor performance on certain benchmarks is how the subproblems are produced. For instance, if we use existentially quantified variables to produce subproblems, but all subproblems are satisfiable, this results in each *PaQuBE* client needlessly searching a satisfiable subproblem, when only one satisfiable subproblem needs to be searched. Thankfully, with intelligent conflict clause and solution cube sharing, each *PaQuBE* client can still learn from one another, thus minimising this redundant work. As a side note, if we focus on splitting variables and speedup, universal splitting variables tend to do better (2.45x for $\forall$ versus 1.62 for $\exists$).

Now, with respect to knowledge sharing, we show how many clauses (*#CCS* & *#CCT*) or cubes (*#SCS* & *#SCT*) are shared and taken in total (rather than per second), and how many actually produced implications (*#Impl.*) or conflicts (*#Confl.*) on average per client. It is easy to see that only a small fraction of clauses and cubes seem to be directly useful. This information highlights the need for better heuristics. Again, solution cube sharing seems to perform better than conflict clause sharing with respect to how many clauses are shared/taken, but again this is benchmark family dependant. However, as can be seen from Table 1, conflict clause sharing seems to reduce the total run time more. Also, certain benchmarks share only conflict clauses, while other only share cubes. This is because certain problems consist of mostly solution space searches, while other are conflict based searches. The main reason the adaptive algorithms perform so well, is due to this. Basically, there is a huge difference between the structures of each benchmark family.

Lastly, the performance of our compression algorithm for solution cubes is shown (*Comp.*). Here we see that on average our solution cubes are compressed by over $15\times$. This reduces the size of the packets we need to send, and therefore the total network congestion. This should help in the future as we scale to larger systems.

6 Conclusion and Future Work

In this paper we introduced the parallel QBF solver *PaQuBE*. It is based on the state-of-the-art QBF solver *QuBE*, which according to the last QBF competition is significantly faster than other sequential solvers. We then highlighted some of the problems associated with static knowledge sharing criteria, and proposed some better adaptive methods. We also introduced a new compression algorithm that quickly compresses cubes by over $15\times$ on average. Finally, as with many research endevors, we introduced new questions. For example, while our adaptive knowledge sharing strategy performs well, it's still not optimal.

Lastly, we plan to test *PaQuBE* on a larger cluster, currently being installed at the University of Genova. This cluster will contain multiple, multicore IBM servers connected by an Infiniband network (20Gb/s) with over 40 processors in total when installed later this year.

Acknowledgments

The authors would like to thank the German DAAD and the Italian AIT for their support (Vigoni). Furthermore, this work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative

Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS). See www.avacs.org for more information.

References

[1] M. Benedetti. skizzo: A suite to evaluate and certify qbfs. In *Proc. CADE*, pages 369–376, 2005.

[2] A. Biere. Resolve and expand. In *Proc. SAT*, pages 59–70, 2004.

[3] W. Chrabakh and R. Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[4] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem proving. *Communication of ACM*, 5(7):394–397, 1962.

[5] N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with qbf. In *SAT*, pages 408–414, 2005.

[6] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate Quantified Boolean Formulae. In *Proc. AAAI*, 2000.

[7] I. P. Gent and A. G. Rowley. Solution learning and solution directed backjumping revisited. Technical Report APES-80-2004, APES Research Group, February 2004. Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.

[8] L. Gil, P. Flores, and L. Silveira. Pmsat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.

[9] E. Giunchiglia, P. Marin, and M. Narizzano. *Reasoning with Quantified Boolean Formulas*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 761–780. IOS Press, February 2009.

[10] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.

[11] E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE++: An efficient QBF solver. In *5th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004*, pages 201–213, 2004.

[12] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *Journal of Artificial Intelligence Research (JAIR)*, 26:371–416, 2006.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[14] Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel sat solving. In C. Boutilier, editor, *IJCAI*, pages 499–504, 2009.

[15] M. Herbstritt and B. Becker. On Combining 01X-Logic and QBF. In *Proceedings of 11th International Conference on Computer Aided Systems Theory (EuroCAST)*, pages 531–538, Las Palmas de Gran Canaria, Canary Islands, Spain, 2007. Springer Verlag.

[16] H. Kleine-Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.

[17] M. Lewis, P. Marin, T. Schubert, M. Narizzano, B. Becker, and E. Giunchiglia. Paqube: Distributed qbf solving with advanced knowledge sharing. In *Proceedings of SAT, International Conference on Theory and Applications of Satisfiability Testing*. Springer Verlag, 2009. LNCS.

[18] M. Lewis, T. Schubert, and B. Becker. QMiraXT – A Multithreaded QBF Solver. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, January 2009.

[19] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[20] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995.

[21] Y. Yu and S. Malik. Validating the result of a quantified boolean formula (qbf) solver: theory and practice. In T.-A. Tang, editor, *ASP-DAC*, pages 1047–1051. ACM Press, 2005.

[22] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, 1996.

[23] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 200–215, 2002.

Solver	#CCS/s	(size)	#CCT/s	(size)	#SCS/s	(size)	#SCT/s	(size)	#PS	Time	Time Red.
QuBE 1P	–	(–)	–	(–)	–	(–)	–	(–)	231	59,398.23	1.00
PaQuBE 4P-NOS	–	(–)	–	(–)	–	(–)	–	(–)	263	36,263.55	1.64
PaQuBE 4P-CS1	21.13	(17.85)	0.83	(16.44)	–	(–)	–	(–)	265	36,651.70	1.62
PaQuBE 4P-CS2	25.05	(27.15)	1.08	(27.66	–	(–)	–	(–)	264	36,112.51	1.64
PaQuBE 4P-CS3	23.05	(35.33)	1.03	(33.33	–	(–)	–	(–)	267	36,414.35	1.63
PaQuBE 4P-CSADP	23.64	(33.57)	1.03	(33.45)	–	(–)	–	(–)	268	35,671.97	1.67
PaQuBE 4P-SS1	–	(–)	–	(–)	13.81	(14.88)	2.71	(9.27)	264	36,638.83	1.62
PaQuBE 4P-SS2	–	(–)	–	(–)	13.91	(18.07)	3.12	(8.670	264	36,844.03	1.61
PaQuBE 4P-SSQA	–	(–)	–	(–)	107.01	(101.09)	4.58	(20.11)	265	36,997.55	1.61
PaQuBE 4P-SSFU	–	(–)	–	(–)	6.46	(72.23)	3.39	(6.51)	266	36,850.64	1.61
PaQuBE 4P-SSADP	–	(–)	–	(–)	124.06	(142.19)	5.36	(29.68)	267	35,550.72	1.67
PaQuBE 4P-ver-[17]	27.00	(37.02)	1.19	(17.96)	22.77	(27.43)	4.29	(8.56)	268	34,916.82	1.70
PaQuBE 4P-CSSSADP	24.44	(36.08)	1.02	(54.17)	126.87	(142.4)	5.48	(30.92)	270	34,287.86	1.73

Table 1: PaQuBE: Comparison Of Knowledge Sharing Strategies

Family	Insts.	4P ×	Conflict Clauses				Solution Cubes				
			#CCS	#CCT	#Impl.	#Confl.	#SCS	#SCT	#Impl.	#Confl.	Comp.
Abduction	13	2.19	26,280	621.50	0.17	0.83	187,670	1,869.00	0.00	0.00	10.31
BMC	12	0.91	467,060	19,816.67	290.00	198.33	43,420	12,371.50	0.00	0.17	5.55
Cond. Pl.	2	1.70	6,610	300.83	0.33	0.50	0	0.00	0.00	0.00	NA
counter	1	1.50	6,010	198.33	0.17	0.50	3,870	491.00	0.00	0.00	14.86
Ev-Pr-*-lg	7	11.97	1,610	55.83	0.00	0.33	2,280	769.33	0.00	0.00	61.03
FPGA_PFS	1	8.53	2,510	65.50	0.00	0.00	0	0.00	0.00	0.00	NA
irqlkeapcite	1	4.00	0	0.00	0.00	0.00	2,130	3.67	0.00	0.00	4.33
k_*_n	16	1.25	5,250	208.33	3.00	13.17	21,370	1,663.83	0.00	8.00	12.13
k_*_p	18	1.17	16,500	683.00	0.00	0.33	17,000	5,926.17	0.33	76.67	9.67
katz	2	15.98	8,300	66.00	0.00	0.00	1,330	414.50	3.83	0.00	1.86
logn	1	1.45	680	27.33	1.00	1.33	0	0.00	0.00	0.00	NA
sakallah	43	1.35	4,500	156.50	0.00	0.00	1,583,010	25,293.00	207.83	57.67	26.08
Scholl-Becker	7	1.39	64,260	2,018.00	0.00	0.00	23,440	3,372.00	0.00	0.00	4.15
Sorting Networks	15	2.50	80,920	3,797.00	0.00	0.00	106,650	86,137.00	73.17	82.33	2.73
Szymanski	4	1.28	0	0.00	0.00	0.00	2,380	1,773.33	0.00	0.00	0.97
terminator	21	2.50	20	1.00	0.00	0.00	346,750	11,594.33	0.83	11.50	3.31
tipfixdiameter	32	2.46	88,320	4,247.00	0.17	7.33	406,400	22,827.83	3.83	602.50	9.64
tipfixpoint	79	1.46	56,220	2,633.00	0.50	2.33	1,581,040	13,357.67	0.33	43.33	7.34
TOILET	5	9.02	2,970	70.33	0.00	0.00	0	0.00	0.00	0.00	NA
wmiforward	3	4.43	0	0.00	0.00	0.00	21,530	99.00	0.00	0.33	3.69
Total	283	1.73	838,020	34,966.17	295.33	225.00	4,350,270	187,963.17	290.17	882.50	15.33

Table 2: PaQuBE: Benchmark Family Performance