Clause Sharing and Partitioning for Cloud-Based SMT Solving

Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina

Università della Svizzera italiana, Switzerland

Abstract. Satisfiability modulo theories (SMT) allows the modeling and solving of constraint problems arising from practical domains by combining well-engineered and powerful solvers for propositional satisfiability with expressive, domain-specific background theories in a natural way. The increasing popularity of SMT as a modelling approach means that the SMT solvers need to handle increasingly complex problem instances. This paper studies how SMT solvers can use cloud computing to scale to challenging problems through sharing of learned information in the form of clauses with approaches based on both divide-and-conquer and algorithm portfolios. Our initial experiments, executed on the OpenSMT2 solver, show that parallelization with clause sharing speeds up the solving of instances, on average, by a factor of four or five depending on the problem domain.

1 Introduction

The *Satisfiability Modulo Theories* (SMT) [5] approach to constraint solving consists of determining whether a logical formula is satisfiable, given that some of the Boolean variables have an interpretation in *background theories*. The expressiveness of SMT makes it suitable for a vast range of application domains, including software and hardware model checking [9,4], bioinformatics [26], and optimization [24], and has recently attained significant interest from a wide range of users. The computational cost of solving SMT instances can be very high, given that already propositional satisfiability is an NP-complete problem and the introduction of background theories can only make the problem harder. The SMT solvers tackle complexity with a tightly integrated loop where the SAT solver attempts to find a satisfying solution and queries the validity of a candidate solution from the theory solvers. In case the candidate solution is shown to be invalid the theory solvers and the SAT solver work together to extract new expressive constraints in the form of learned clauses, by combining theory specific information and resolution.

This work studies how employing parallelism and in particular cloud computing can be used in helping SMT solvers to scale to increasingly hard problems. We study two different approaches: a portfolio where several copies of a randomized SMT solver is run on a single instance; and an approach where the SMT instance is divided into several partitions that are guaranteed by construction not to share models, and each partition is solved by an SMT solver. We combine

these two approaches in a natural way by having several SMT solvers work on each partition. The emphasis of the work is in how the different ways of organizing the search can co-operate to speed up the solving. We implement the co-operation by having SMT solvers working on the same partition share the clauses they learn during the execution.

To study the effects of parallelism and clause sharing we implement the approach using the SMT solver OpenSMT2 [13] and experiment with two particularly central background theories, the quantifier-free theories of uninterpreted functions and equalities [5] (QF_UF) and linear real arithmetics [7] (QF_LRA). The experimental results suggest that both the portfolio and the partitioning based approach can greatly benefit from clause sharing. Interestingly, a comparison between portfolio and partitioning reveals that the portfolio approach performs better even if the partitioning is combined with portfolio. We give an analysis in the form of a case study to understand the reason for this and confirm the effect in a more controlled experiment. Finally we discuss to what extent the results obtained with OpenSMT2 can be generalized to other SMT solvers. In particular clause sharing with partitioning is tedious to implement in a solver and therefore we make the comparison in an indirect way, studying the run-time distribution of the Yices2 solver [6] in comparison to OpenSMT2.

Related Work. The portfolio approach combined with clause sharing has been implemented using the SMT solver Z3 [25]. The implementation provides an efficient clause sharing strategy within the same computer using lockless queues that hold references to the lemmas that a solver core wants to export. The experimental evaluations show that clause sharing leads to a substantial speedup on benchmarks from the QF_IDL logic. In contrast to this work, we support two SMT theories (QF_UF and QF_LRA), and exploit the advantages of combining portfolio with search-space partitioning. Moreover our implementation is designed to run in a cluster or a cloud in addition to a single machine. Similarly to Z3, the SMT solver CVC4 [3] supports a portfolio-style parallel solving. Unlike our approach, the approach used in CVC4 is designed to run in a single computer and does not implement clause sharing.

In [14] we introduced the parallelization tree formalism for combining portfolio and search-space partitioning. The work also describes and reports results on the QF_UF logic on some instantiation of the framework. Our work extends this tool based on the OpenSMT2 SMT solver by introducing clause sharing and the logic QF_LRA.

A divide-and-conquer approach for the quantifier-free bit-vector logic has been implemented on top of the SMT solver Boolector [23]. A portfolio parallelization approach for the logic of quantifier-free bit-vectors and bit-vector arrays is presented in [21]. Compared to these, our work differs in the supported theories and in that we support cloud computing and are not limited to pure divide-and-conquer or portfolio approach.

In this work we use techniques similar to those used in parallel SAT solving. The more elaborate problem descriptions of SMT constitute a significant theoretical and engineering challenge for parallelization. In addition the use of SMT

allows extending these techniques to a different domain. While the results are to some extent preliminary it seems already that there are substantial differences in how the techniques perform in the two domains. Given the close relation of the topics there is a significant amount of relevant research on parallel SAT solving, overviewed for instance in [17]. In particular we point out the portfolio approach combined with clause sharing implemented in ManySAT [8] and HordeSAT [2]. A promising future direction for SMT is the combination of search-space partitioning and clause sharing [1].

Recently there has been a renewed interest in parallel model checking. In [22], the authors give a method for parallel concolic execution, while [10] introduces a method for using massive parallelism to obtain a high coverage in an explicit-state model-checking approach in a stochastic way. These differ from our work in that they do not provide solutions directly for SMT solving.

In this paper we first introduce the basic concepts required for interpreting our results in Sec. 2, and then describe implementation details in Sec. 3. The experimental results obtained with the implementation are presented in Sec. 4, and conclusions are drawn in Sec. 5.

2 Background

This section gives an overview of how SMT solvers work concentrating on the mechanisms that are relevant for interpreting the framework, implementation, and experimental results we present in the following sections. In describing the preliminaries we use the set notation.

A literal is a Boolean variable x or its negation $\neg x$. A clause is a set of literals and a propositional formula in conjunctive normal form (CNF) is a set of clauses. Throughout the text we use both a set of literals and disjunction, and a set of clauses and a conjunction, interchangeably. An assignment σ is a set of literals such that for no variable x, both $x \in \sigma$ and $\neg x \in \sigma$. A variable x is *assigned* if either $x \in \sigma$ or $\neg x \in \sigma$. An assignment σ satisfies a clause c if $\sigma \cap c \neq \emptyset$ and a formula F if it satisfies all its clauses.

Most SMT solvers are based on the DPLL(T) framework [20] which takes as input a problem instance presented as a propositional formula where some of the Boolean variables have an interpretation as Boolean relations, such as equalities, disequalities, and inequalities, in a theory T. A DPLL(T) solver consists of a solver for the propositional satisfiability problem (SAT) and one or more theory solvers that are capable of reasoning on a conjunction of Boolean relations over the theory T. In the pre-processing phase the input formula is converted into an equisatisfiable propositional formula F in CNF while preserving the special T-interpretations of the Boolean variables.

The SMT solving process is driven by a SAT solver maintaining a set of clauses which initially consists of the formula F. During the search the SAT solver builds an assignment σ and alternates between two phases.

- In the *propagation* phase the solver identifies clauses $c = l_1 \vee \ldots \vee l_n$ such that (i) there is a single unassigned literal $l_i \in c$, and (ii) σ falsifies the literals

$l_j, 1 \leq j \leq n, j \neq i$. Any such literal l_i is added to σ until no new clauses satisfying (i) and (ii) can be found.
- In the *decision* phase the solver chooses a literal l_i unassigned in σ and adds it to σ.

A *conflict* occurs if during the propagation phase the SAT solver detects a clause with all literals falsified. The decisions and propagations are stored in the *implication graph* [16], a directed graph having as nodes the literals of the assignment σ and as edges the arches $\{(\neg l_j, l_i) \mid 1 \leq j \leq n, j \neq i\}$ obtained in the propagation phase.[1]

A SAT solver *learns* a clause c by performing essentially resolution steps directed by the implication graph when it finds a conflict. The learned clauses are by construction guaranteed to be logical consequences of F, and are both used in guiding the search and added temporarily to the clause database to reduce the number of assignments the solver needs to cover during the search. Finally the solver makes with a decreasing frequency a *restart* where the assignment σ is cleared and the search is continued without otherwise changing the state of the solver.

The SAT solver queries periodically whether the conjunction of the theory atoms in σ is consistent with the theory. In case a theory solver determines an inconsistency it identifies a subset $\sigma' \subseteq \sigma$ that causes the inconsistency and returns the clause $c_T := \{\neg l \mid l \in \sigma'\}$ to the SAT solver. Minimizing σ' is critical for the good performance of the SMT solvers (see, e.g., [19,7]). The clause c_T is used together with the implication graph to learn a clause c in the way described above for clause learning. The solving process terminates when either the clause database becomes unsatisfiable or a satisfying assignment consistent with the theory T is found.

Parallel algorithm portfolios. An algorithm portfolio [11] is a set of algorithms that compete in finding a solution for a given problem. The decision phase employs an heuristic for choosing l_i and introduces in a natural way nondeterminism into the solver. Small changes to the heuristic can cause big changes in the run time of the solver. For example, Fig. 5 shows the effect of allowing the SAT solver to make random choices against the heuristic in small number of cases to a single instance. The lines labeled OpenSMT2 and Yices2 illustrate the probability of solving an instance from the QF_LRA category of the SMT-LIB benchmark collection in a given time or number of decisions for the SMT solvers OpenSMT2 and Yices2 [6], respectively. A natural algorithm portfolio can be obtained by seeding differently the pseudo-random-number generator of the SMT solver and running several solvers in parallel.

Clause sharing. In clause sharing the clauses learned by an SMT solver while solving a formula F are distributed among the solvers in the parallel portfolio. Since clause learning plays an important part of the SMT solving process this sharing can speed up the parallel solving process. For example the shared clauses

[1] We equate x and $\neg\neg x$.

make it easier to produce the required clauses in case of unsatisfiability, and reduce the number of assignments the solver covers before finding a satisfying assignment.

Search space partitioning. The SMT solver bases its search on the SAT solver, and therefore a natural way of dividing the work and avoiding overlap for the solvers is to constrain the SMT formula F into partitions $F_1, \ldots, F_n$ such that the original formula is satisfiable if and only if one of $F_1, \ldots, F_n$ is satisfiable. This can be done through adding additional constraints C_i to the formula F, resulting in the partition $F_i := F \wedge C_i$. In principle conjoining a single literal $\{l\}$ to the formula F halves the search space, but this happens rarely in practice. Often the resulting partitions F_i will have overlap in their search due to the heuristics of the solver and therefore the observed speedup will be less dramatic. The situation is made worse by the unpredictability of the SMT solver run time in case the instance is unsatisfiable. Assuming the shape of the run-time distribution is the same for both the instance F and the partition F_i, it can be shown that independent of the number of partitions there are distributions for which the expected run time increases when partitioning is done as described above [15]. To lessen this effect several more complex parallelization algorithms combining elements from search-space partitioning and algorithm portfolios have been suggested [14]. For example running a parallel portfolio for each partition makes it less likely that one of the partitions will require excessive time for being solved.

Constructing partitions. We use in this work an approach for constructing partitions called *scattering*, initially introduced in [12]. Given a formula F, a number of partitions to be created n, and a sequence of positive $d_1, \ldots, d_{n-1}$ the partitions are obtained following the iteration

$$
\begin{aligned}
F_1 &:= F \wedge\ l_1^1 \wedge \ldots \wedge l_{d_1}^1 \\
F_k &:= F \wedge\ (\neg l_1^1 \vee \ldots \vee \neg l_{d_1}^1) \wedge \ldots \wedge (\neg l_1^{k-1} \vee \ldots \vee \neg l_{d_{k-1}}^{k-1}) \wedge l_1^k \wedge \ldots \wedge l_{d_k}^k \\
F_n &:= F \wedge\ (\neg l_1^1 \vee \ldots \vee \neg l_{d_1}^1) \wedge \ldots \wedge (\neg l_1^{n-1} \vee \ldots \vee \neg l_{d_{n-1}}^{n-1})
\end{aligned}
$$

The goal of using the sequence d_i is to make the search space of each F_i as close as possible to $1/n$ of the search space of the instance F. We obtain the sequence d_i by assuming that conjoining a disjunction of k literals with a formula F reduces the size of the search space by factor of $(1 - 1/2^k)$. For example for $n = 2$ partitions this gives $d_1 = 1$, while for $n = 8$ we get the sequence $d_1 = 3, d_2 = 3, d_3 = 3, d_4 = 3, d_5 = 2, d_6 = 2$, and $d_7 = 1$. Note that the number of constructed partitions in this method does not have to be a power of 2. Finally, the literals l_i^j are chosen using the same heuristic the SMT solver uses during the search.

3 The parallelization Framework

In this section we present the framework uses in the experiments of the paper for studying the effect of parallelization approaches and clause sharing. We give

an overview of the framework in Fig. 1. The framework is designed to run on a cluster of computers or a cloud, even though it is also possible to run the system on a single computer. The design follows a client-server approach in which the server, acting as a front-end to the user, receives input instances in the SMTLIB2 format[2], and at the same time handles the connection with the clients, managing client failures and asynchronous new client connections gracefully. The clients are implemented as SMT solvers wrapped by a network layer that handles the connection with the server.

The server works in two modes: depending on the configuration it either splits the instance into several partitions using the scattering approach described in Sec. 2, or runs in a pure portfolio mode without splitting. In the beginning of the solving the server distributes either the partitions or the original formula to the solvers in all the available clients.

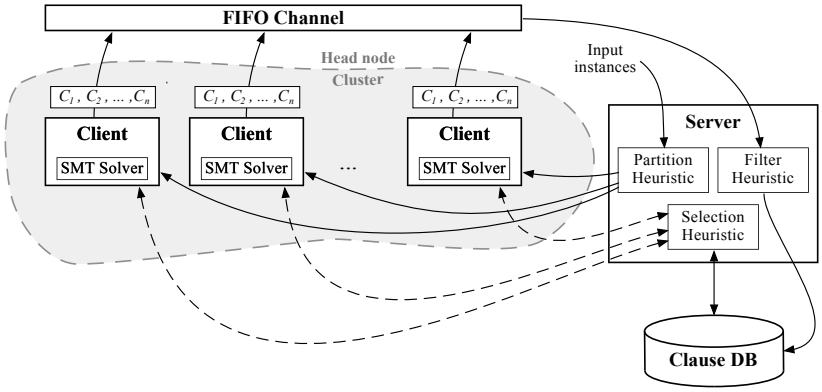

Fig. 1. Parallel SMT solver framework with clause sharing

During the solving phase, each client contacts the server both to publish newly learned clauses and to request new interesting clauses published by the clients. To avoid problems with the high throughput we use a *FIFO Channel* that allows multiple clients to push clauses to the server without turnaround delays. Once a batch of clauses is received, the server uses the *filter heuristic* to choose potential clauses for merging them with the previously received clauses into the clause database (*Clause DB* in Fig. 1). Hence, at any given time, that database will contain all the learned clauses sent by the clients and that have passed the filter heuristic of the server. Inside the clause database the clauses are divided by partitions and each client has only access to the clauses published by the solvers working on the same partition.

[2] http://smtlib.cs.uiowa.edu

Upon a restart each client will ask for clauses from the server. However, the request cannot be replied by sending the entire clause set of the partition that is being solved, because it usually consists of prohibitively many clauses. A high number of clauses slows down the client solver since the overhead related to the growth of the internal data structure is higher than the speedup obtained from the clauses. To address the problem, the framework allows the use of the *selection heuristic* which attempts to choose clauses that are particularly promising for the problem at hand. The current version implements naïve heuristics for both filtering and selection. The filtering heuristic is based on the number of literals inside each clause: clauses with more than a fixed number of literals will be discarded. The selection heuristic works by randomly selecting a fixed number of clauses from the database, and each new set will replace the old one inside the solver.

3.1 Implementation

The goal of the implementation is to provide scalability, fault-tolerance, and low latencies during data transfer, as well as ease of use and portability from a cluster of machines to a single machine with many cores or CPUs. In the rest of this section we will present the central choices made during the implementation. The implementation is highly modular in order to allow studying the effects of its components in isolation. For example the system can handle several different policies for scheduling the partitions among the solvers in the cluster. This allows research on how to best combine portfolio and search-space partitioning.

The SMT solver. The abstract framework allows the use of any DPLL(T) solver, but our current implementation uses the SMT solver OpenSMT2 [13]. OpenSMT2 is a light-weight SMT solver that currently supports the quantifier-free theories of uninterpreted functions with equalities (QF_UF) and linear real arithmetics (QF_LRA). The solver is written in C++ and has been developed in Università della Svizzera italiana since 2008. The code is easily approachable because of its limited size of roughly 50,000 lines of code and the object-oriented architecture. In addition it is released under the MIT license[3]. Most recently the solver competed in the SMT competitions in 2014 and 2015, performing in the mid-range in the competition. The implementation is efficient featuring low-level memory management and a cache-friendly design for many of the central algorithms. These reasons make it our choice over other, maybe more optimised tools.

Networking. The server and the clients communicate using our custom-built message passing protocol through TCP/IP sockets, making the solution light-weight, easy to implement and modify as well as portable by being compatible with clusters, cloud computing and single computers. The network components of the implementation are shown in Fig. 1. Almost all the connections consist of

[3] http://opensource.org/licenses/MIT

push message passing which achieves the goal to be as fast as possible by avoiding the turnaround time. The connection between solvers and the selection heuristic is the only one that does not use the push mechanism, but instead needs a pull request. This choice has been made since the schedule at which the clients can receive clauses is unpredictable, making push impractical. To indicate the pull request, the interaction is drawn with dashed lines.

A binary format for SMT. In order to use clause sharing we must ensure that the internal clausal representation of each instance is the same in every client solver. This property cannot be guaranteed by the SMTLIB2 language since small changes in the input formula might result in subtle optimizations that will dramatically change the CNF structure seen by the SAT solver embedded in the SMT solver. For that reason we designed a binary format for SMT, representing the internal state of OpenSMT2. This format is used for data transfers between each client and the server but also results in us being forced to limit ourselves to a specific SMT solver.

FIFO channel and clause DB. For these challenges we use REDIS[4], an open source in-memory data structure store, used as database, cache, and message broker. In order to get a scalable, fast, and fault-tolerant push connection from multiple sources we use the *Publisher / Subscriber messaging paradigm* of REDIS. The clauses are stored using the REDIS *SET* feature that automatically handles cases where a clause would be added to clause database that is syntactically equal to an already present clause. The SET feature is used by both the filter and the selection heuristics.

4 Experiments

This section describes the experiments we performed on the implementation described in the paper. The implementation is available at `http://verify.inf.usi.ch/opensmt`. The experiments concentrate on four topics: Sec. 4.1 demonstrates how the clause sharing works on the (i) pure portfolio and (ii) the approach where we split the instance into partitions and use a portfolio for solving each partition; Sec. 4.2 studies the difference between the approaches (i) and (ii) above; Sec. 4.3 reports how the filtering heuristic affects the performance of the algorithm; and Sec. 4.4 compares the cloud-based implementation against a sequential version of OpenSMT2 and a widely used reference solver Z3 [18].

Our hardware configuration is kept the same in all experiments we run. The experiments were run in a cluster where we used eight compute nodes for the clients and the head node for the server. Each compute node is equipped with two CPU Quad-Core AMD Opteron 2384 and 16GB of RAM. During the experiments each cluster node had eight client processes implementing the SMT solver OpenSMT2, resulting in total of 64 solvers in the entire cluster. We did not explicitly limit the memory available to the solvers. The timeout is fixed

[4] http://redis.io

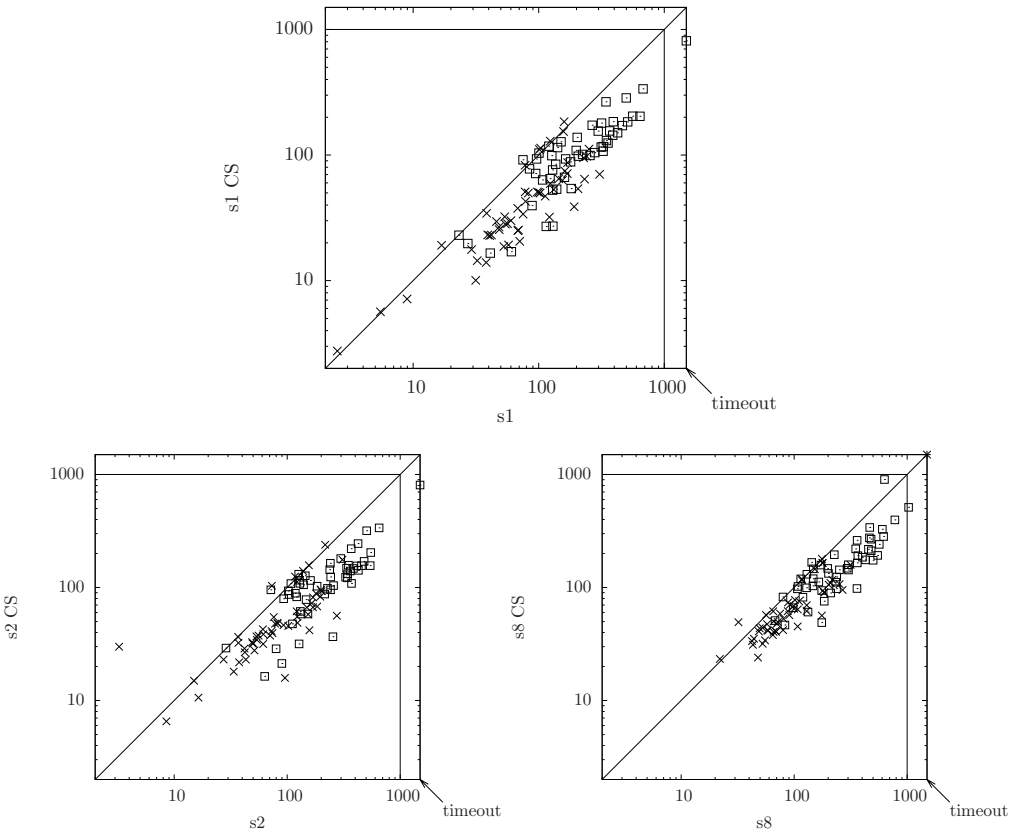

Fig. 2. Using clause sharing against not using clause sharing with 1, 2, and 8 partitions. Framework run with 64 solvers on the QF_LRA benchmark set

everywhere to 1000 seconds. The search-space partitioning heuristic, when used, is the scattering approach [12].

We used a fixed benchmark set obtained from the SMTLIB2 benchmarks repository[5] and the QF_LRA and QF_UF theories. The set from the QF_LRA theory was created by selecting the instances with an average sequential execution time between 100 and 1000 seconds (including those in timeout) using OpenSMT2; the set consists of 106 instances in total. The benchmark set for the QF_UF theory consists of 254 instances. This set includes all instances that could be solved with the sequential OpenSMT2 between 100 and 1000 seconds; 11 instances which are known to be difficult for OpenSMT2 and time out in 1000 seconds; and 200 randomly chosen instances of which half are guaranteed to be satisfiable and the other half unsatisfiable.

[5] http://smtlib.cs.uiowa.edu/benchmarks.shtml

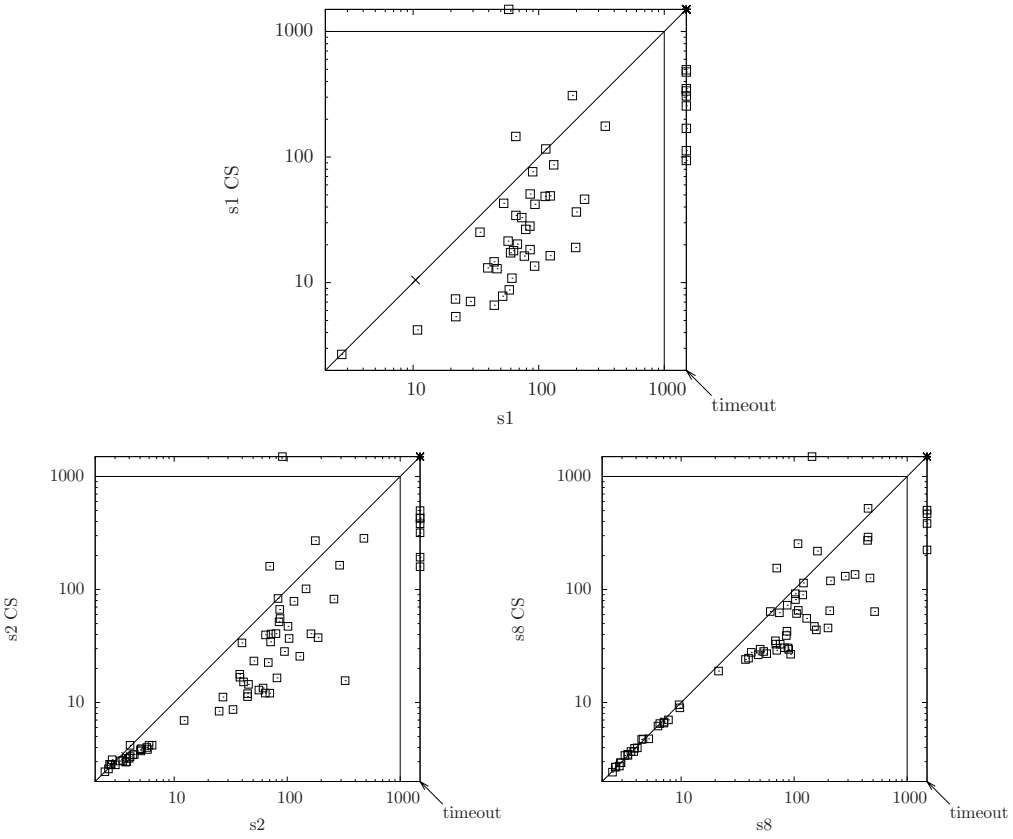

Fig. 3. Using clause sharing against not using clause sharing with 1, 2, and 8 partitions. Framework run with 64 solver on the QF_UF benchmark set

In the figures we use the labels *S1, S2* and *S8* to indicate the number of partitions created from the input instance. Therefore the label *S1* indicates the pure portfolio approach. The label *CS* indicates that clause sharing is used. Throughout the plots we denote satisfiable instances with the symbol $\times$ and unsatisfiable instances with the symbol $\square$.

4.1 The Effect of Clause Sharing

Our first experiments show how sharing clauses affects the solving time using different partitioning methods for QF_LRA (Fig. 2) and QF_UF (Fig. 3). For both figures the graph on the top shows how the parallelization algorithm based on pure portfolio benefits most from clause sharing: with both theories it gives a 2.05 times speedup, as well as one more QF_LRA instance and nine more QF_UF instances solved within the timeout compared to not using clause sharing.

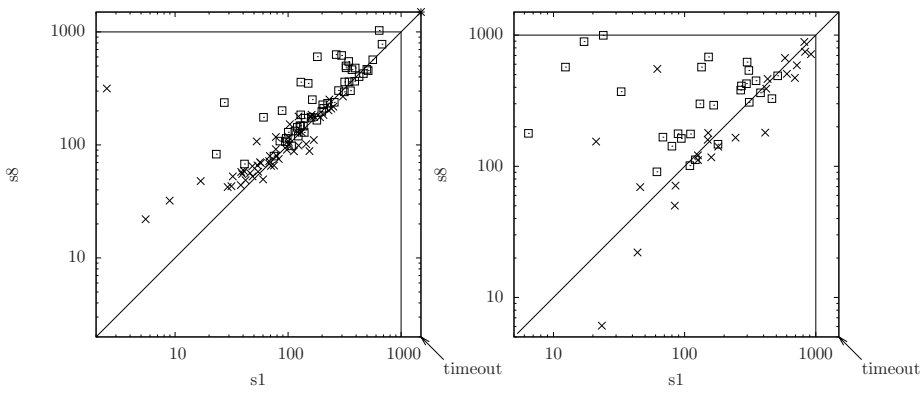

Fig. 4. Comparing *S1* against *S8* on the QF_LRA benchmark set. The graph on the left shows the results using the framework of Sec. 3, the graph on the right shows a controlled experiment where network delays are removed.

With both theories the combination of portfolio and search-space partitioning performs worse than pure portfolio: the speedup due to clause sharing is 1.97 times for partitioning in two and solving each partition with a portfolio of 32 solvers (*S2*), and speedup of 1.67 for partitioning in eight and solving each partition with a portfolio of eight solvers (*S8*).

To some extent these results are expected, since the number of learned clauses available inside the clause database for a single portfolio is bigger when there are more solvers running in the portfolios, and therefore also the quality of clauses that the heuristic picks is higher.

4.2 The Effect of Partitioning

Figure 4 (*left*) compares the portfolio approach against the approach where an instance is split into eight partitions in the framework, from the QF_LRA benchmark set. Interestingly the portfolio approach is almost consistently better than the approach using partitioning, in particular for the unsatisfiable instances (denoted with □). To study this effect in more detail and to rule out effects such as network delays or time used in constructing the partitions, we designed a second experiment in more controlled setting (Fig. 4 (*right*)). For this experiment we chose a set of instances that require more than 1000 seconds to solve using the sequential version of OpenSMT2. The instances were split off-line into eight partitions and each partition was solved with a portfolio of eight OpenSMT2s to obtain the results for the vertical axis. The horizontal axis shows the minimum solving time over 64 OpenSMT2s. The benchmark sets are different on the two figures.

The more controlled experiment verifies the phenomenon that an approach based on partitioning performs worse in particular in the unsatisfiable instances,

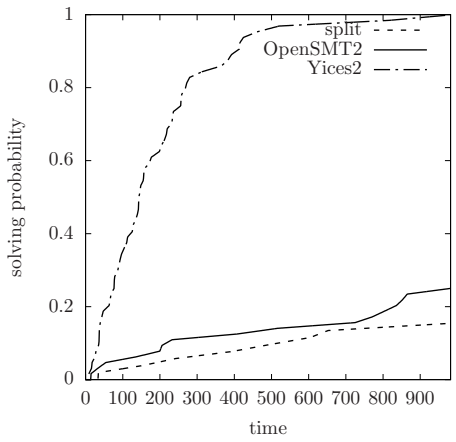

Fig. 5. Run time distribution for solving an example instance with a single solver compared to the distribution of one of the splits. For reference the figure also shows the decision distribution for the Yices2 SMT solver on the original problem (the scale is not shown).

while the results seem to be better for many of the satisfiable instances. This behavior is often observed when the shape of the distribution of an unsatisfiable instance has most of the probability mass at relatively low run times but still a significant mass at significantly higher run time [15]. In such cases the effect of partitioning that tends to make instances easier to solve is not enough to compensate the benefit from a pure portfolio approach. To further study this phenomenon we chose one of the instances where this effect was particularly pronounced, and constructed the run time distribution for OpenSMT2 for this instance and a partition that was empirically difficult. The results for this experiment are reported in Fig. 5. First, the run time distribution shows that there is only a 25% probability that OpenSMT2 solves this instance within the timeout of 1000 seconds, even though the fastest run time for this instance is only slightly above 10 seconds. This explains the good behavior of the portfolio approach. Second, based on the experiment it is possible that the partition run time is in fact higher than that of the original instance run time. The difference is not big and therefore this could be an effect caused by low amount of samples (64 in total). Finally, to understand to what extent this phenomenon is generalizable to other SMT solvers we ran the original instance 64 times using a randomized version of the SMT solver Yices2. Instead of reporting the run-time distribution, we show the number of decisions Yices2 did on this problem, since the run times were too low to get meaningful results. We can see that also for Yices2 the amount of decisions needed varies greatly but the shape of the distribution seems to be different. The observation that the run time with another SMT solver is much faster suggests that this instance can be solved by using an

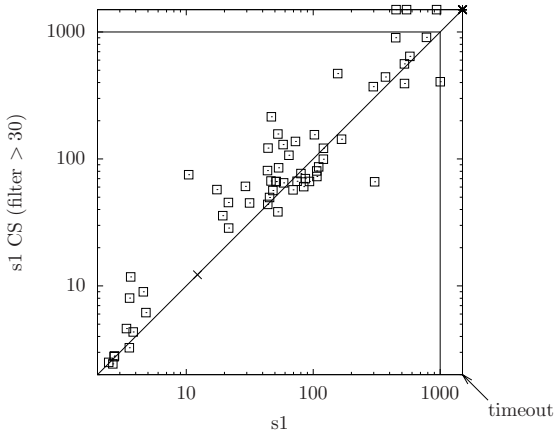

Fig. 6. Using clause sharing with a loose filtering heuristic against not using clause sharing. Framework run with 64 solvers on the QF_UF benchmark set.

optimization that is not implemented on OpenSMT2 and that in such cases it is not safe to draw the conclusion that the partitioning approach would not work well if this optimization were implemented in the underlying solver.

4.3 The Clause Sharing Heuristics

Fig. 6 shows that clause sharing heuristics are very important: the experiment performed using a filtering heuristic that discards clauses with more that 30 literals results in clause sharing having 1.12 times greater run time compared to the run without clause sharing. Interestingly the same heuristic is working well for QF_LRA (used in Fig. 2). To obtain good results for our benchmark instances in QF_UF the heuristic needs to be more restrictive. Reducing the threshold to 10 literals still leads to worse performance (results not shown), and discarding clauses with five or more literals gives the results on Fig 3.

4.4 Comparison to Other Solvers

Figure 7 compares the best known configuration of the framework against the solvers OpenSMT2 and Z3 for QF_LRA (*top*) and for QF_UF (*bottom*). The results are very promising when compared to OpenSMT2. For instances with sequential run time higher than one second and for which neither the sequential or the parallel solver timed out the average case speed-up is 4.78 for QF_LRA and 4.01 for QF_UF. Our implementation is not yet competitive against Z3 in the majority of instances. This is due to the lack of optimizations in the underlying solver. Based on the experimental evidence presented in this section it seems reasonable that if either the optimizations available in Z3 were implemented in

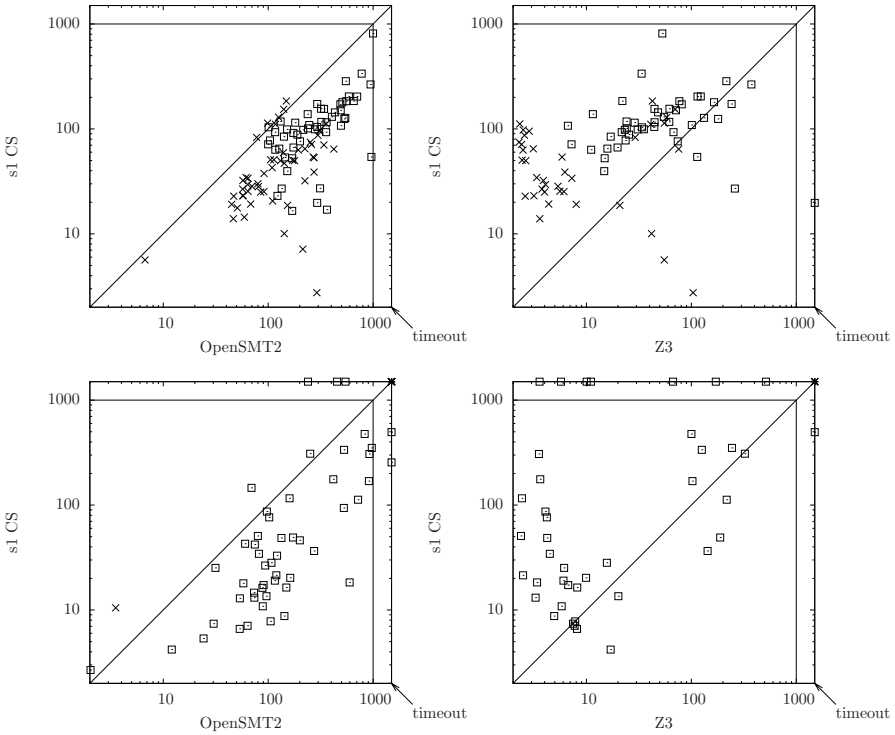

Fig. 7. Our best configuration against OpenSMT2 and Z3 for QF_LRA (*top*) and QF_UF(*bottom*)

OpenSMT2 or the approach presented in this work were implemented in Z3 the results would be similarly promising in comparison to Z3.

5 Conclusions

SMT solving in cloud environments so far has received relatively little attention from the community developing and researching SMT solvers. This paper addresses the challenges related to integrating one of the key components of a modern SMT solver, the sharing of learned clauses, to parallel SMT solving algorithms. We provide a generic framework for clause sharing in a cloud computing environment and implement a system that supports clause sharing with parallelization algorithms based on both a portfolio and splitting the input formula into partitions.

The framework and the parallelization algorithms are agnostic to the underlying theories used by the SMT solver. We provide results on two fundamental theories: the quantifier-free theories of uninterpreted functions and equality

(QF_UF), and linear real arithmetics (QF_LRA). The results show that both theories can benefit significantly from clause sharing, but especially QF_UF is sensitive to the heuristic used for selecting clauses to be shared. In the experiments we also observe that the partitioning approach, while working relatively well for QF_UF, performs somewhat worse for QF_LRA in the benchmark set we study on this paper. We conjecture that this results from the partitioning heuristic behaving in an unexpected way where the problems sometimes get more difficult to solve, in combination with a run-time increasing phenomenon observed with unsatisfiable instances.

Finally we address the question of how well the parallel computing results obtained with one solver generalize to other solvers. Experimentally we observe that the run-time distribution of an instance, one of the key factors determining how parallelization works on an instance, can be dramatically different on two solvers. Therefore it is difficult to estimate the speed-up of a given instance on one solver based on results from another solver. It is nevertheless likely that observations made in this paper would carry over to other solvers in general.

Future work. The framework we set in this paper opens several interesting research directions. In particular we point out two central open questions we plan to address in the future: how to construct a good heuristic for (i) partitioning for QF_LRA and (ii) for filtering and selecting the clauses to be shared in a portfolio.

Acknowledgements. This work was financially supported by SNF project number 200021_153402.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Piette, C.: Dolius: A distributed parallel SAT solving framework. In: Berre, D.L. (ed.) POS-14. EPiC Series, vol. 27, pp. 1–11. EasyChair (2014)
2. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Proc. SAT 2015. LNCS, vol. 9340, pp. 156–172. Springer (2015), `http://dx.doi.org/10.1007/978-3-319-24318-4_12`
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV 2011. LNCS, vol. 6806, pp. 171 – 177. Springer (2011)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS '99. LNCS, vol. 1579, pp. 193–207. Springer (1999)
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
6. Dutertre, B.: Yices 2.2. In: CAV 2014. LNCS, vol. 8599, pp. 737 – 744. Springer (2014)
7. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proc. CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer (2006)
8. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability Boolean Modeling and Computation 6(4), 245 – 262 (2009)

9. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In: Tenth International Workshop on Model Checking of Software (SPIN). LNCS, vol. 2648, pp. 235–239. Springer (2003)
10. Holzmann, G.J.: Cloud-based verification of concurrent software. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 311–327 (2016), http://dx.doi.org/10.1007/978-3-662-49122-5_15
11. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
12. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Proc. SAT 2006. LNCS, vol. 4121, pp. 430–435. Springer (2006)
13. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Proc. SAT 2016. pp. 547 – 553. No. 9710 in LNCS, Springer (2016)
14. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Proc. SAT 2015. LNCS, vol. 9340, pp. 369–386. Springer (2015)
15. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundamenta Informaticae 107(2-3), 289–311 (2011)
16. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
17. Martins, R., Manquinho, V.M., Lynce, I.: An overview of parallel SAT solving. Constraints 17(3), 304–347 (2012)
18. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337 – 340. Springer (2008)
19. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Proc. RTA 2005. LNCS, vol. 3467, pp. 453 – 468. Springer (2005)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937 – 977 (2006)
21. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proc. CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer (2013)
22. Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S.: Parallel SMT solving and concurrent symbolic execution. In: 2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3. pp. 17–26 (2015), http://dx.doi.org/10.1109/Trustcom.2015.608
23. Reisenberger, C.: PBoolector: a Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead. Master's thesis, Johannes Kepler Univesität Linz, Linz, Austria (2014)
24. Sebastiani, R., Tomasi, S.: Optimization modulo theories with linear rational costs. ACM Transactions on Computational Logic 16(2), 12:1–12:43 (2015)
25. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: A concurrent portfolio approach to SMT solving. In: Proc. CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer (2009)
26. Yordanov, B., Wintersteiger, C.M., Hamadi, Y., Kugler, H.: SMT-based analysis of biological computation. In: Proc. NFM 2013. LNCS, vol. 7871, pp. 78–92. Springer (2013)