Search-Space Partitioning for Parallelizing SMT Solvers

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

Faculty of Informatics, University of Lugano
Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland

Abstract. This paper studies how parallel computing can be used to reduce the time required to solve instances of the Satisfiability Modulo Theories problem (SMT). We address the problem in two orthogonal ways: (i) by distributing the computation using algorithm portfolios, search space partitioning techniques, and their combinations; and (ii) by studying the effect of partitioning heuristics, and in particular the lookahead heuristic, to the efficiency of the partitioning. We implemented the approaches in the OpenSMT2 solver and experimented with the QF_UF theory on a computing cloud. The results show a consistent speed-up on hard instances with up to an order of magnitude run time reduction and more instances being solved within the timeout compared to the sequential implementation.

1 Introduction

The *Satisfiability Modulo Theories* problem [9,27] (SMT) is the problem of determining whether a propositional formula is satisfiable, given that some of the Boolean variables have an interpretation as equalities or inequalities in a background theory. The problem has recently gained importance as a modeling approach for a vast range of application domains from software model checking (see, for instance, [12,1]) to optimization [4,21,30,26] due to its expressiveness and the efficient implementations. One of the features that make the SMT framework inviting for domain specialists is its flexibility in admitting a wide range of theories. The theory of quantifier-free uninterpreted functions with equalities (QF_UF) [9] is one of the most fundamental and applicable background theories, being widely used in combination with other theories (for a list of SMT theories see `http://www.smt-lib.org/`), and for instance as an abstraction to obtain more efficient decision procedure implementations [6]. The computational cost of solving SMT instances can be very high, given that already determining the propositional satisfiability is an NP-complete problem and the introduction of background theories can only make the problem harder. Nevertheless there has been relatively little research on how parallel computing can be used to speed up the solving of SMT instances (see the related work section for a survey).

This work addresses the challenge of parallelization. We introduce an abstract parallel algorithmic framework for SMT called the *parallelization tree*. The framework allows combining two important approaches for parallelization:

algorithm portfolios and the divide-and-conquer approach. The key idea of the framework is that both solving and partitioning the search space can be done with a portfolio. The approach is applicable to all SMT solvers based on the DPLL(T) paradigm independent of the used theories, and our experiments address the central QF_UF theory. We show experimentally, both with a parallel solver implemented in a cloud computing environment and with an experimentation in a more controlled environment, that several instantiations of the parallelization tree framework are very efficient in solving SMT instances. We are able to solve more instances within a given timeout, observe sometimes an order of magnitude speed-ups, and are competitive with optimized SMT solvers on hard instances.

Most SMT solvers, including [3,10,7,24,6], consist of a SAT solver that searches for a satisfying assignment for a problem instance represented as a set of clauses, and theory solvers that check whether the assignment is consistent with respect to the theory in question. The SAT solver finds a satisfying assignment for its clause set, and the theory solvers check the consistency of the set of Boolean variables that have an interpretation in the theories. The found inconsistencies are communicated to the SAT solver as clauses that the solver adds as refinements to its clause set. The process terminates once the SAT solver has found a satisfying assignment consistent with the theories, or when the theory solvers have provided enough clauses for the SAT solver to determine unsatisfiability.

One of the challenges in parallelizing SMT solvers using the divide-and-conquer approach is that the clause set of the SAT solver does not initially contain the full information on the SMT instance unlike in SAT solving. As a result the approaches for parallelizing SAT solvers are not directly applicable to SMT solving. Our approach addresses this challenge in two ways. For constructing partitions we develop versions of the lookahead and the VSIDS heuristic [23] that are both made aware of the theory solver. The parallelization tree approach, on the other hand, is used to increase the probability of quick solving through the use of portfolios both for solving instances and constructing partitions. To the best of our knowledge the applications of the parallelization tree framework, partitioning, and lookahead in SMT with QF_UF are all new.

Related Work. The portfolio approach for parallel SMT solving is studied in [31] for problems from the QF_IDL logic. The system, implemented in the Z3 SMT solver, provides an efficient clause-sharing strategy for the workers and concludes that the best results are obtained with a random portfolio similar to ours. In this work we use instead the QF_UF logic, study different types of parallelization approaches, and scale the solver to more CPUs. A divide-and-conquer approach for the QF_BV logic is studied in [29]. The procedure tries to solve the formula with the Boolector SMT solver within a given timeout. If no result is obtained within the timeout the formula is divided into two partitions using a heuristic based on lookahead and the search is continued in parallel on the resulting partitions. The procedure terminates when Boolector returns a satisfiable result for one formula, or all formulas are shown unsatisfiable. We also study the lookahead heuristic for constructing partitions but concentrate on the QF_UF logic and provide a more

general parallelization algorithm. A portfolio-style parallelization approach for the QF_ABV logic is presented in [28]. While the work uses several SMT solvers as the portfolio, in our work we concentrate on implementing all the parallelization approaches inside a single solver.

There is a substantial body of recent work on parallel SAT solving, and for instance [22] gives a good overview of the recent advances. A number of future challenges in parallel SAT solving in particular and in parallel constraint programming in general is given in [13]. Our work discusses in part the challenge of combining the portfolio style search and the divide-and-conquer approach, a topic we believe to be orthogonal to the ones presented in [13]. More recently [2] presents an approach based on search space partitioning, both with and without clause sharing. A theoretically oriented study in [20] suggests that parallelizing a SAT solver using a portfolio might in some cases be inherently difficult because of the resolution structure. We seem to observe a similar barrier with SMT solvers, and show experimentally that the divide-and-conquer approach, when used in combination with a portfolio, seems to overcome this problem. The lookahead heuristic [15] has been previously used in SAT solving and proved particularly effective in constructing partitions [18,14]. In this work we extend this line of work and use for the first time lookahead for parallelizing SMT solvers with QF_UF.

2 Preliminaries

Given a finite set of Boolean variables $B = \{x_1, \ldots, x_n\}$, a *clause* is a set of *literals*, that is, positive and negative Boolean variables $x, \neg x$, $x \in B$. A *propositional formula in conjunctive normal form* (CNF) is a conjunction of clauses. In the context of this work an *SMT formula* F is a propositional formula given in CNF where the variables of a subset $B_T \subseteq B$ have an interpretation as equalities over terms of a theory T. If $x \in B_T$, the literal $\neg x$ is interpreted as the corresponding disequality in the theory T. In this work we will consider the quantifier free theory of uninterpreted functions with equalities (QF_UF).

A *truth assignment* $\sigma \subseteq \{x, \neg x \mid x \in B\}$ is a set of literals such that for no $x \in B$ both $x \in \sigma$ and $\neg x \in \sigma$. A truth assignment is *total* if for all $x \in B$ either $x \in \sigma$ or $\neg x \in \sigma$. A clause c is *propositionally satisfied* if $\sigma \cap c \neq \emptyset$ and a CNF formula F is propositionally satisfied if all its clauses are propositionally satisfied. The formula F is *satisfiable* if there is an assignment σ satisfying propositionally F, and the set of equalities and disequalities imposed by the assignment σ on the equalities in B_T interpreted in the theory T is consistent. A literal l is *implied* under σ in F if there is a clause $c \in F$ such that $l \in c$, $l \notin \sigma$ and for all other $l' \in c, l' \neq l$ it holds that $\neg l' \in \sigma$.[1] A truth assignment σ is *conflicting* if for some variable x either both x and $\neg x$ are implied or for some $l \in \sigma$, $\neg l$ is implied.

An *SMT solver* consists of a SAT solver and a theory solver that communicate by exchanging equalities and negations of equalities from the set B_T as clauses.

[1] We follow the convention that $\neg\neg x = x$ for $x \in B$

The SAT solver starts with an initial set of clauses F. The solver communicates periodically non-conflicting assignments to a theory solver. Upon receiving an assignment σ, the theory solver interprets the set $\sigma_T = \sigma \cap \{x, \neg x \mid x \in B_T\}$ as equalities and disequalities and determines whether σ_T is inconsistent or consistent with the theory T. In case of consistency the theory solver may provide the SAT solver with a set of *theory-implied literals* $l \notin \sigma$. If the theory solver agrees on the consistency of a total truth assignment σ, the assignment σ is returned as a proof of satisfiability for F. If the theory solver finds theory-implied literals these are communicated to the SAT solver as clauses that imply the theory-implied literals. For any theory-inconsistent truth assignment the theory solver will communicate a set of *theory clauses* consisting of variables in B_T and expressing the reason for the inconsistency of the assignment. During the search the SAT solver can find *learned clauses*, that is, clauses that the SAT solver has derived using its current clause database with conflict analysis based on resolution. A learned clause c_l has the property that if F is the current set of clauses, then any truth assignment σ propositionally satisfying F also propositionally satisfies c_l. In contrast a theory clause learned right after communicating the satisfying truth assignment σ for F is not propositionally satisfied by σ. Finally, the *unit propagation closure* $U(F, \sigma)$ is the smallest set of literals containing σ that is closed under a rule that includes to σ all implied and theory-implied literals.

3 Parallelization Approaches for SMT

Heuristics for guiding the search on a boolean structure play an important role in both SAT and SMT solvers. As a natural consequence of the computational difficulty of the SMT problem heuristics are inaccurate and small changes can result in significant differences in run times. This phenomenon can be used to obtain speed-up in a parallel setting using a *portfolio* of algorithms. The main challenge in parallelizing SMT solvers this way is that portfolio-style solving seems to hit a scalability limit where adding more CPUs does not provide more speed-up [18,19,20]. The scalability problem of the portfolio-style solving can be addressed by allowing the search processes to share information such as learned and theory clauses. The approach has been studied for SMT in [31] where it was shown that sharing both types of clauses helps speeding up the solver. In this work we use the simple portfolio obtained by forcing the SAT solver to make certain choices randomly, potentially against the heuristic values. This approach was found to be efficient in SAT solving [17] and was identified to be the best-performing strategy for SMT solvers in [31].

However, this paper targets also the scalability limit in an orthogonal way by using a divide-and-conquer approach where several solvers work in parallel on problem instances that are constructed by partitioning the search-space of the original instance and hence are different from each other. The solution to the original problem instance can be obtained by combining the results from the partitioned instances. This approach has an inherent problem that needs to

be addressed to obtain good results: If the original problem instance is unsatisfiable, the variance in run time results in decreased performance. Instead of having to solve a single instance the solver needs to solve several instances that, despite being usually easier than the original, might still be challenging. While the variance in solver run time makes the portfolio approach efficient, it degrades the performance of the divide-and-conquer approach, effectively resulting in the solver having to wait for the "unluckiest" instance to be solved. Under certain assumptions it can be shown that for implementations based on pure divide-and-conquer it is possible to come up with a run-time behavior that results in increased run time when parallelized this way, and that a different organization of the search can help to avoid this problem [19].

We show experimentally that often the partitions constructed from the original problem are somewhat easier but not significantly so, and this results in such slowdown anomalies. However, even in cases where the instances do not get significantly easier it is possible to obtain good speed-up by using a portfolio approach on both constructing and solving the partitions. To present our approach we will formalize the idea of combining divide-and-conquer with portfolio. We introduce an abstract parallelization algorithm framework called *parallelization tree* and give five concrete instantiations of the framework. In addition to providing us with a convenient tool for discussing different parallelization algorithms the framework is also used as a tool for explaining the performance results we present in Sec. 5. We will introduce an even more practical implementation of the framework in Sec. 4.2 which uses also a load balancing schema.

In the following we first discuss certain approaches for partitioning the search-space in SMT and then describe the parallelization tree framework. We conclude with concrete examples of the framework.

3.1 Search-Space Partitioning in SMT

The basic approach for constructing partitions in SMT uses a *partitioning function*, denoted by $partf_n : F \mapsto F_1, \ldots, F_n$, to divide an SMT instance F into n partitions $F_1, \ldots, F_n$. The function satisfies the conditions that F is satisfiable if and only if $F_1 \vee \ldots \vee F_n$ is satisfiable and no two partitions F_i, F_j, $i \neq j$, share a satisfying truth assignment. We construct partitions by conjoining *partitioning constraints* $P_1, \ldots, P_n$, in general set of clauses, to F. We use two types of partitioning constraints: the ones obtained with the *scattering approach* [18] and the ones obtained with *guiding paths* [5,32].

The scattering approach. The scattering approach is a technique for partitioning an instance into arbitrary number of partitions. Each partitioning constraint P_i is obtained by heuristically selecting a set of *scattering literals* $l_1^i, \ldots, l_{k_i}^i$ and conjoining these literals with the clauses obtained by negations of the previous scattering literals. More formally this can expressed as $P_i := l_1^i \wedge \ldots \wedge l_{k_i}^i \wedge (\neg l_1^{i-1} \vee \ldots \vee \neg l_{k_{i-1}}^{i-1}) \wedge \ldots \wedge (\neg l_1^1 \vee \ldots \vee \neg l_{k_1}^1)$. The number of scattering literals k_i are selected so that the partitions constructed are approximately equally sized under the assumption that fixing a literal will reduce the search space with a

constant factor. If n is the number of partitions to be generated, it can be shown that the fraction obtained by fixing the scattering literals $l_1^i, \ldots, l_{k_i}^i$ should be $r_i = \frac{1}{n-i+1}$ of the previous instance F_{i-1} [16]. We simply assume that fixing a literal will half the search space, and this results in us choosing the number of scattering literals k_i minimizing the difference $|r_i - 2^{-k_i}|$.

The guiding paths. As an alternative to the scattering based method for constructing the partitions we use a simple variant of the *guiding path* approach [32] where a binary tree with literals as nodes represents the $n = 2^k, k \geq 1$ partitions. The root of the tree consists of the *true* literal, and the rest of the nodes are either leaves with no children or have exactly two children v and $\neg v$ where $v \in B$. Each path $\mathit{true}, l_1, \ldots, l_k$ from the root to a leaf l_k corresponds to a partitioning constraint $l_1 \wedge \ldots \wedge l_k$.

3.2 Combining Search Space Partitioning and Portfolio

The key idea in obtaining well-performing parallel solvers where search-space partitioning plays a role is to combine elements from both the search-space partitioning and the algorithm portfolio.

The *parallelization tree* abstract algorithmic framework provides a unified way of presenting and comparing different parallelization algorithms. The parallelization tree consists of two types of nodes: *and-nodes* and *or-nodes*. The root and the leaves of the parallelization tree are and-nodes. Each and-node is associated with an SMT instance and, with the possible exception of the root of the parallelization tree, with one or more SMT solvers. The instance at the root of the parallelization tree is satisfiable if any instance in the and-nodes is shown satisfiable. A subtree rooted at an and-node is unsatisfiable if one of its children is unsatisfiable or at least one of the solvers associated with the and-node has shown the instance unsatisfiable. A tree rooted at an or-node is unsatisfiable if every tree rooted at its children is unsatisfiable.

We use a *partitioning operator* $\mathrm{split}^k(n_1, \ldots, n_k, F)$ to construct the parallelization tree. The result of applying the operator split^k on an and-node F is a tree rooted at the and-node F having k children $o_1, \ldots, o_k$. Each child node o_i is an or-node and has as children the and-nodes $a_1^i, \ldots, a_{n_i}^i$. Finally, each and-node a_j^i is associated with the partition obtained by applying the (randomized) partitioning function partf_{n_i} on the formula F.

As instances of the parallelization tree we identify five particularly interesting parallelization algorithms.

- The *plain* partitioning approach $\mathit{plain}(n, F)$ corresponds to the parallelization tree $\mathrm{split}^1(n, F)$ where each of the instances associated with the nodes $a_1^1, \ldots, a_n^1$ is solved with a single SMT solver.
- The *portfolio* approach $\mathit{portf}(k, F)$ corresponds to the parallelization tree consisting of the root associated with the instance F and using k SMT solvers to solve the instance.

– The *safe* partitioning approach $\mathit{safe}(n, s, F)$ corresponds to the parallelization tree $\mathrm{split}^1(n, F)$ and solving each of the instances $a_1^1, \ldots, a_n^1$ with s SMT solvers.
– The *repeated* partitioning approach $\mathit{rep}(n, k, F)$ corresponds to the parallelization tree $\mathrm{split}^k(n, \ldots, n, F)$ where each instance associated with the nodes $a_1^1, \ldots, a_n^1, \ldots, a_1^k, \ldots, a_n^k$ is solved with one SMT solver.
– The *iterative* partitioning approach $\mathit{iter}(k, F)$ corresponds to the infinite parallelization tree where every instance associated with an and-node is being solved with a single SMT solver and every and-node associated with an instance F_a has the single or-child and and-grandchildren constructed by applying the operator $\mathrm{split}^1(n, F_a)$.

Figure 1 illustrates the corresponding parallelization trees and the solver assignments. When clear from the context, we omit the formula F as well as the other parameters from the partitioning approach.

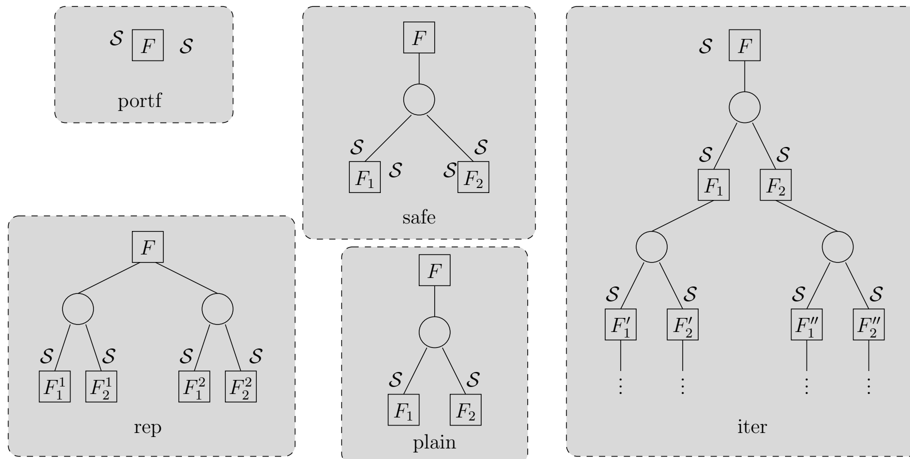

Fig. 1. Example parallelization trees (clockwise from the top left): $\mathit{portf}(2, F)$, $\mathit{safe}(2, 2, F)$, $\mathit{iter}(2, F)$, $\mathit{plain}(2, F)$, and $\mathit{rep}(2, 2, F)$. The and-nodes are drawn with boxes, and the or-nodes with circles. The SMT solvers are indicated with the symbol $\mathcal{S}$.

Concrete SMT instantiations of the parallelization tree include the CVC4 and Z3 SMT solvers which implement a portfolio, and PBoolector [29] which implements an iterative partitioning approach.

4 A Cloud-Based Parallel SMT Solver for QF_UF

We have implemented the approaches discussed in this work into the OpenSMT2 solver. The solver is a complete rewrite of the SMT solver OpenSMT [6] and includes all the algorithmic optimizations present in OpenSMT. However, due to improved memory management, reduced memory footprint of the critical data

structures, and certain bug fixes, the current version is approximately 10% faster on the QF_UF family of benchmarks.

The SAT solver inside OpenSMT2 is based on MiniSAT 2.0 [11], a conflict-driven clause-learning SAT solver. The congruence closure algorithm implemented in OpenSMT2 employs the algorithm from [25] for communicating small reasons for unsatisfiability of equalities containing uninterpreted functions to the SAT solver.

To be able to distribute in an unambiguous way the partitioning constraints and partitions to the parallel working solvers we have implemented a format where the propositional encoding into CNF is made explicit. Due to the format we are able to do the partitioning in a more general way, using also Boolean variables created with the Tseitin transformation that are not in general part of the original problem description. The format consists of the CNF corresponding to the Tseitin encoding of the SMT instance, and the learned and the theory clauses available at the point when the output is constructed. The format also contains the sorts and the terms defined in the input instance, and the mapping between the terms and the Boolean variables.

In the following we describe details related to the implementation of the solver: the heuristics used for constructing partitions, and the architecture of the cloud-based tool.

4.1 Partitioning Heuristics

We implemented two different heuristics for the partitioning approach: the VSIDS-based heuristic [23] which scores higher the variables that are often involved in conflicts, and the lookahead heuristic which will give high scores to variables that propagate a high number of literals. The VSIDS heuristic is used together with the scattering approach for constructing partitions, while the lookahead heuristic is used with the simplified guiding path approach. When using the VSIDS heuristic we dedicate a short amount of time to perform a search on the instance so that the VSIDS heuristic gets reasonable scores for the variables.

The lookahead heuristic starts with an assignment σ and computes for each variable $x \notin \sigma$ the sizes of the sets $U(F, \sigma \cup \{x\})$ and $U(F, \sigma \cup \{\neg x\})$. The highest score is assigned to the variable that maximizes the minimum of the sizes of these two sets. As a result the heuristic favors variables that construct similar sized partitions having few variables. We have implemented a few important optimizations for the approach: if a literal l propagates n literals $l_1, \ldots, l_n$, the heuristic sets the number n for the upper bounds for all literals $l_1, \ldots, l_n$. If a variable is propagated both in the positive and in the negative polarity, and if the lower of the upper bounds is lower than the current best value, the lookahead on this variable can be safely skipped. Additionally if the literal propagation results in a conflict this means that the solver can learn an arbitrary number of learned and theory clauses and the current heuristic values are no longer valid. In this case the heuristic does not restart the lookahead from the beginning, but continues instead from the next literal. This trick is known to reduce the run time of the lookahead in practice by a linear factor.

4.2 The Client-Server Architecture

We implemented a subset of the functionality of the parallelization tree framework into the OpenSMT2 solver designed to run in a computing cloud or a cluster. The tool is available at `http://verify.inf.usi.ch/parallel-opensmt2/`.

The system follows the client-server architecture where the server receives a set of SMT instances and an arbitrary number of connections from clients. The server then manages the construction of partitions from the original instances and distributes the partitions to the clients for solving. The server and the clients communicate using our custom-built protocol through TCP/IP sockets, making the solution light-weight, portable, easy to modify and easy to use.

The clients are implemented as processes with two threads, one communicating with the server and the other responsible for the solving of the SMT instance. The communicating thread waits for an instance from the server, passes the instance to the solving thread and then continues to listen to further commands from the server. Currently the supported commands are initiation of a solving of a partition and termination of the solving. If the thread solves the instance before server sends the terminate command the result is communicated to the server and the process returns to the initial state waiting for a new instance from the server. The client is implemented in C++ on top of OpenSMT2, and an architectural overview is given in Fig. 2.

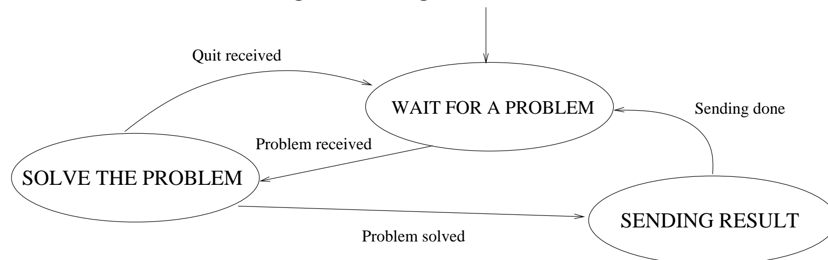

Fig. 2. The client architecture

The server is implemented as a two-threaded Python/2.7 program that calls OpenSMT2 to construct the partitions. Both threads listen to connections on separate TCP ports, the command thread for commands from the user and the worker thread for communication with the clients on solving SMT instances. The worker thread maintains a list of all connected clients and the list of partitions to be solved, and constructs the partitions using the partitioning heuristic and the parallelization tree. In addition the worker thread provides the unsolved partitions to the clients again based on the selected parallelization tree. The command thread communicates with the user. The user may send commands such as initiations of the solving of a new SMT instances, requests to terminate the current solving, or a request to print the status of current jobs. The implementation is capable of handling client failures, exits, and new connections seamlessly and completely automatically. The architecture of the server and the communication between the clients and the user is described in Fig. 3.

Currently the implementation has been adapted to the parallelization algorithm *safe*. The implementation differs from the algorithm discussed in Sec. 3

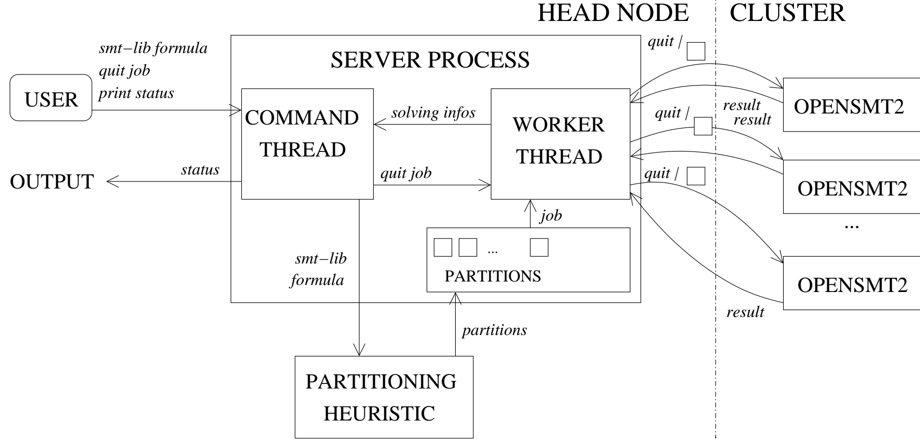

Fig. 3. The Server Architecture

in that it provides load balancing by feeding new unsolved partitions to a client that has shown an instance unsatisfiable. The assigned partition is the one being currently solved by the least number of clients. If there are more than one such partition, one is chosen at random. Despite this difference in the following we will use the notation introduced in Sec. 3 to describe the implementation, but will in addition mention the number of cores used in the computation when relevant.

5 Experimental Results

This section presents the results of some of the algorithms obtained from the *parallelization tree* algorithmic framework, using the cloud-based implementation presented in Sec. 4.2 as a uniform platform for testing. For completeness we also report experimentations against other SMT solvers in Sec. 5.3. The experiment set contains of all instances from the QF_UF category of the SMT-LIB benchmark collection (`http://www.smt-lib.org/`) having run time longer than one minute with the default configuration of OpenSMT2. This set consists of 54 instances, 11 of which could not be solved within the 1000 seconds timeout and 4 GB memory limit. All the instances we could solve from this set turned out to be unsatisfiable, and therefore we also added randomly selected 100 easier satisfiable and unsatisfiable instances, resulting in total 254 benchmark instances. All the experiments were run on a cluster consisting of nodes with two AMD quad-core Opteron 2344 HE CPUs and each node was running at most four solver processes. All times are reported in seconds.

We show the results for the hardest instances in our benchmark set in Table 1. To the table we have selected certain approaches that illustrate the behavior of the parallelization algorithms well. The columns OSMT2^1 and OSMT2^{64} represent, respectively, the sequential run of the OpenSMT2 solver and the run of the cloud-based implementation described in Sec. 4.2, using the VSIDS scattering heuristic, the algorithm $\mathit{safe}(8,8)$ and 64 CPU cores. The rest of the columns correspond to instantiations of the parallelization algorithms discussed in Sec. 3

Table 1. Instances solved with at least one of the approaches, but where the portfolio approach required over 100 seconds with 64 CPUs. All the instances are unsatisfiable.

Name	OSMT2[1]	*portf*(64)	*rep*(2,32)	*safe*(2,32)	*plain*(64)	*rep*(8,8)	*safe*(8,8)	OSMT2[64]
PEQ003_size9	437.37	299.76	336.00	232.81	431.44	286.11	248.32	**195.70**
PEQ004_size8	124.85	117.17	109.15	110.76	125.70	108.07	115.34	**15.85**
PEQ011_size8	572.12	302.12	267.11	265.91	388.54	309.68	280.54	**258.94**
PEQ012_size6	—	—	456.56	507.76	621.24	574.61	532.06	**382.44**
PEQ014_size11	737.58	**338.56**	482.68	564.22	—	—	540.05	539.28
PEQ016_size6	223.68	181.19	158.60	168.57	188.14	158.02	176.94	**20.20**
PEQ018_size7	192.58	144.96	155.99	**139.68**	182.50	207.46	218.02	168.18
PEQ020_size6	511.26	409.50	379.89	**314.26**	405.37	401.27	371.41	337.04
SEQ005_size8	174.85	159.70	144.13	144.28	132.26	148.76	131.84	**16.84**
SEQ010_size8	244.22	190.11	123.36	166.96	196.38	157.59	155.92	160.94
SEQ026_size7	890.18	708.89	731.43	794.43	671.05	774.14	725.79	686.63
SEQ038_size8	—	826.11	903.32	751.73	751.03	745.75	792.07	819.41
NEQ006_size6	—	—	—	—	—	—	—	**16.62**
NEQ016_size8	774.57	616.73	682.64	575.34	—	625.47	**419.60**	592.15
NEQ023_size7	—	—	—	—	—	—	—	**50.75**
NEQ032_size6	—	830.03	407.59	**373.25**	—	836.31	865.95	532.08
NEQ048_size8	476.46	430.31	421.38	**341.27**	479.38	349.94	445.24	458.11
NEQ048_size9	—	815.72	**759.96**	804.73	849.76	832.16	833.86	846.21
Total solved	12	15	16	16	13	15	16	**18**

with scattering and VSIDS heuristic. The reported times include also the time to run the partitioning. The best run time for a given instance is shown in boldface and the dashes indicate timeouts.

The results suggest that for hard instances OSMT2[64] performs very well compared to the other approaches, solving the largest number of instances and usually with a very good run time. The implementation is the fastest solver for eight instances, the runner-up being the parallelization algorithm *safe*$(2, 32)$ with four fastest times. The run time of the implementation is very competitive with the parallelization algorithm *safe*$(8, 8)$. When a formula is partitioned to many sub-instances some of them will be easy. The load-balancing present in the implementation allows the solving approach to concentrate on the hard instances.

The parallelization algorithm *plain* performs badly for our benchmark instances when compared to the other parallelization algorithms. This suggests that often the partitions are not significantly easier. Therefore constructing a large number of partitions with mutually exclusive models easily results in more work for the parallel solver. To illustrate this better we show in Fig. 4 (top left) a comparison between the algorithms *plain*(64) and *rep*$(2, 32)$ using the full benchmark set. The run times include the time required for the partitioning. Here, as well as in all other similar graphs, we denote satisfiable instances by $\times$ and unsatisfiable by $\square$, and highlight timeouts with a red color. We can see that the algorithm *plain*(64) is almost always worse. A closer analysis reveals that when considering only the instances that both approaches could solve the algorithm *rep*$(2, 32)$ solves the full problem set 9 times faster than the algorithm *plain*(64).

Finally we point out that while the parallelization algorithm *portf* works relatively well it seems to loose in the hard instances when compared to the approaches that combine elements from portfolio and search-space partitioning. In fact if we do not consider the sequential execution and the algorithm *plain*, other algorithms perform better than our implementation of the portfolio.

In the following subsections we will study in more detail the observations made based on Table 1, consider role of the heuristic used in constructing the

partitions, and finally conclude with a comparison of our implementation against some other well-known SMT solvers.

5.1 Comparing the Implementation to the Portfolio Approach

Given a fixed amount of parallel CPUs, there is an interesting tradeoff between the number of partitions constructed from an instance and the number of solvers that can be assigned for each partition. The comparison in Table 1 suggests that with this benchmark set and the VSIDS scattering heuristic good performance is obtained by constructing only two partitions and dedicating a large number of solvers for the two partitions (32 in the experiment) when using the parallelization algorithms. The situation changes when we use the parallel implementation with the load-balancing schema since usually some of the constructed partitions are much easier than others, therefore freeing up resources for solving the yet unsolved partitions. Figure 4 compares the implementation of the parallelization algorithms $\mathit{safe}(2, 32)$, $\mathit{safe}(8, 8)$, and $\mathit{safe}(64, 1)$ against the algorithm $\mathit{portf}(64)$ all using 64 cores. The results show that while the easy instances suffer from the overhead caused by the communication in the network and the time required to construct the partitions, the harder instances with run times more than 10 seconds usually profit from the partitioning. The positive effect of partitioning the problem into more than two parts can in particular be seen when comparing the implementations of $\mathit{safe}(2, 32)$ and $\mathit{safe}(8, 8)$. Unlike in the abstract algorithms the implementation with $\mathit{safe}(8, 8)$ performs clearly better than $\mathit{safe}(2, 32)$ solving one more instance and providing a total speed-up of roughly 10% on the instances solved by both approaches. The implementation of $\mathit{plain}(64)$, while still not competitive, also performs significantly better as a result of the load balancing.

5.2 Comparing the Partitioning Heuristics

Since the ability of the partitioning to construct easy instances plays such a critical role in the overall success of the partitioning based instances, it is interesting to study the effect of partitioning heuristics. We compare here two different types of heuristics, the VSIDS scattering and the lookahead with our guiding path implementation. We also experimented with alternations of the VSIDS scattering heuristics that prefer the equalities in the set B_T and the purely Boolean variables $B \setminus B_T$. These however did not result in significant differences in our benchmark set and the results are therefore not shown.

The results for the comparison are given in Fig. 5. Excluding the time to construct the partitions, the lookahead gives a 40% reduction in the run time of the solver when using the abstract algorithms $\mathit{safe}(8, 8)$, the average speed-up being 2. However, when the time required to construct the partitions is included, the lookahead-based heuristic looses the edge and becomes slightly worse compared to the VSIDS-based heuristic. This results mainly from the implementation of the lookahead-heuristic. The current implementation is not as optimized as the

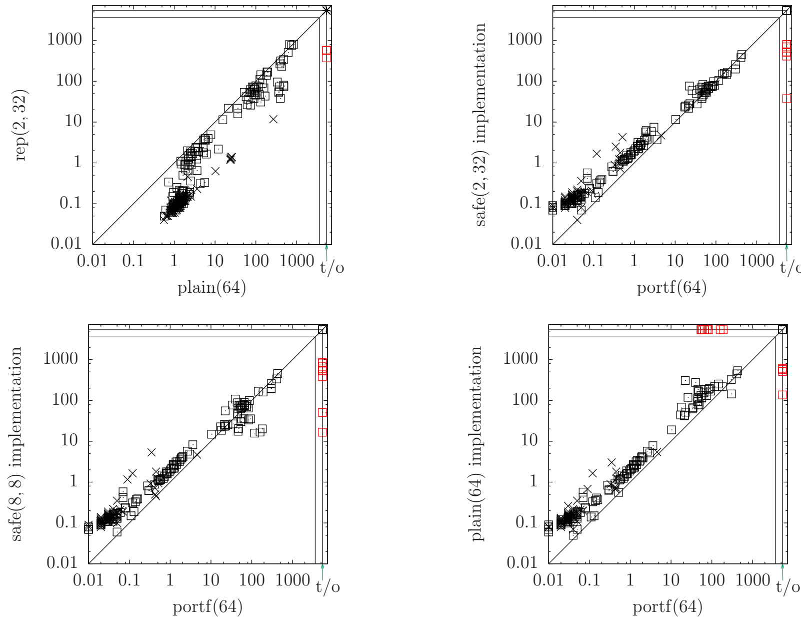

Fig. 4. The run times for the parallelization algorithms *plain*(64) and *rep*(2, 32) (*top left*). The *portf*(64) algorithm compared to the load-balancing implementations of *safe*(2, 32) (*top right*), *safe*(8, 8) (*bottom left*) and *plain*(64) (*bottom right*) on 64 cores.

VSIDS implementation, but we believe that the heuristic can be made more efficient.

To understand the impressive efficiency of the lookahead heuristic we study closer two examples where the abstract parallelization algorithm *safe*(2, 32) performs well with the lookahead heuristic and with the VSIDS heuristic. The graphs on the bottom of Fig. 5 report the cumulative run-time distributions of the original instance and the four partitions constructed with the two heuristics. In the first example the lookahead heuristic finds a partitioning where the two partitions have a very similar run time distribution, whereas the VSIDS heuristic results in a very uneven distribution where one partition is significantly easier to solve than the other. In the second case (lower right graph in Fig. 5) the lookahead heuristic performed on a single run worse than the VSIDS heuristic. In this case both the heuristics resulted in a very uneven partitions. However it would seem that the cumulative run-time distribution of the VSIDS heuristic dominates on a wide area the distribution of the lookahead-based heuristic. Interestingly there seems to be a small probability that the lookahead-heuristic can solve the problem somewhat faster than the original problem, suggesting that the implementation with load balancing should be capable of performing well on this instance also for the lookahead heuristic.

5.3 Comparison to Other SMT Solvers

Finally we report the comparison of the parallel implementation and in particular the implementation of the parallel algorithm *safe*(8, 8) against other solvers

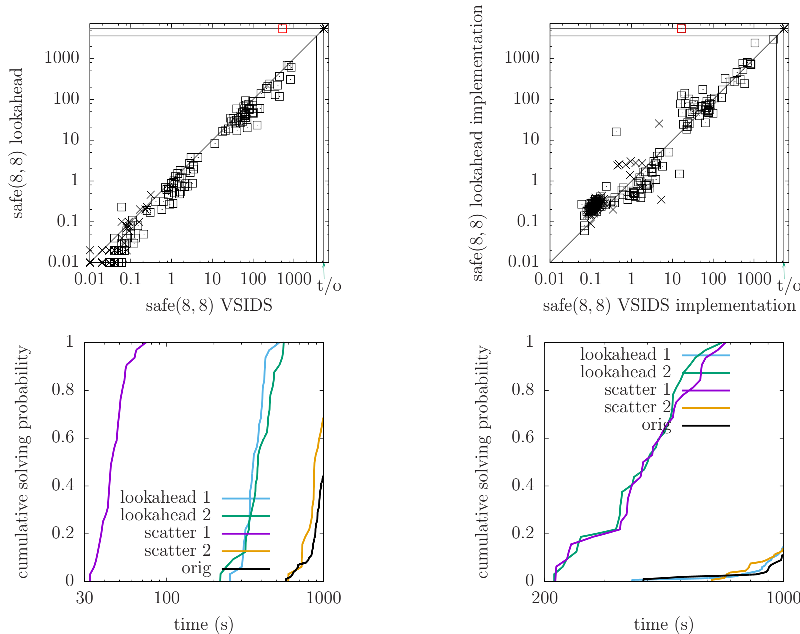

Fig. 5. The lookahead heuristic compared to the VSIDS-based scattering heuristic

in Fig. 6. All solvers were run with the default configurations. We first note that the implementation provides a clear speed-up against the sequential version of OpenSMT2, being 75% faster in the total run time over the benchmark set and solving six more instances within the timeout. However, the parallel implementation suffers a penalty related to the communication delays and constructing partitions when the instances are easy. The comparison against MathSAT5 [7] shows similar behavior: the parallel implementation can solve a handful of instances within the timeout that MathSAT5 could not solve. Comparing the solver against CVC4 [3] reveals that the parallel implementation is capable of solving nine more instances, using in total 12% less wall-clock time for solving all instances in the benchmark set. Nevertheless there are several instances that CVC4 solves much faster than the parallel implementation. We believe the reason for this is a symmetry breaking simplification [8] implemented in CVC4 that is particularly effective on some of the benchmarks in our set. Finally the comparison to Z3 [24] shows that even though the parallel implementation of OpenSMT2 is not yet competitive, there are some instances we could solve from the benchmark set that Z3 could not solve and several others where it is likely that the parallelization results in much lower run times compared to Z3. For lack of space we need to omit the comparison to certain other solvers such as Yices2 [10]. We believe that due to optimizations the results of the comparison would be similar to that of Z3.

6 Conclusions

Approaches for solving unsatisfiable constraint problem instances based on purely divide-and-conquer suffer from the phenomenon that an inefficient partitioning

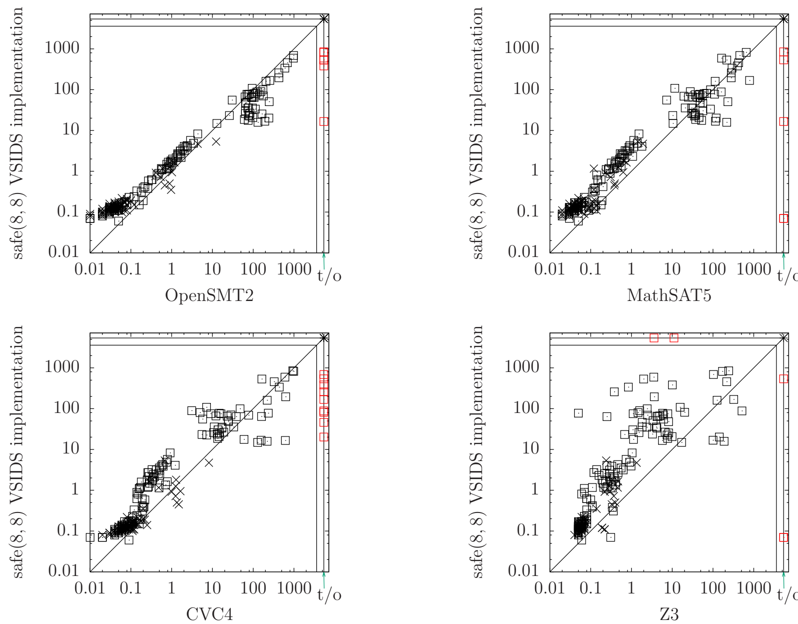

Fig. 6. Comparison of the *safe*$(8, 8)$ implementation with the scattering heuristic against other SMT solvers.

results in several instances that are roughly as difficult to solve as the original instance. As a result it is common to use a portfolio of different solvers to overcome this problem. This paper presents the generic framework called parallelization tree for combining the portfolio approach with partitioning. We present how several parallel algorithms can be seen as instances of this framework, and provide implementations for some of the parallel algorithms for the SMT problem with the logic QF_UF in computing cloud. We show with a thorough experimentation that the implementations provide a significant speed-up, and are capable of solving several more instances within a given time-out compared to the sequential implementation. Furthermore we show that the implementations are competitive against many state-of-the-art SMT solvers.

Based on the results we are able to point out certain directions for future research. We believe that there is still work to be done in the heuristics for constructing partitions: our implementation of the lookahead heuristic is fairly straightforward and there are several techniques that can be used to improve its performance. One such technique is identifying equalities and inequalities of the variables. Also generalizing the lookahead to a portfolio in the way it was done for the VSIDS heuristic seems like a viable alternative for obtaining efficient partitionings. Finally we are interested in applying the knowledge obtained in this study to a setting where we allow the parallel-running solvers to exchange also learned and theory clauses.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This work was financially supported by SNF project number 153402.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Proc. CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer (2012)
2. 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)
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. Bjørner, N., Phan, A., Fleckenstein, L.: νz - an optimizing SMT solver. In: Proc. TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer (2015)
5. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver: Efficient workload balancing. Annals of Mathematics and Artificial Intelligence 17(4–3), 381–400 (1996)
6. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Proc. TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer (2010)
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS 2013. LNCS, vol. 7795, pp. 93 – 107. Springer (2013)
8. Déharbe, D., Fontaine, P., Merz, S., Paleo, B.W.: Exploiting symmetry in SMT problems. In: Proc. CADE-13. LNCS, vol. 6803, pp. 222–236. Springer (2011)
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
10. Dutertre, B.: Yices 2.2. In: CAV 2014. LNCS, vol. 8599, pp. 737 – 744. Springer (2014)
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'03. LNCS, vol. 2919, pp. 502–518. Springer (2004)
12. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: Proc. IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer (2010)
13. Hamadi, Y., Wintersteiger, C.M.: Seven challenges in parallel SAT solving. AI Magazine 34(2), 99–106 (2013)
14. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Proc. HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer (2011)
15. Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 155–184. IOS Press (2009)
16. Hyvärinen, A.E.J.: Grid-Based Propositional Satisfiability Solving. Ph.D. thesis, Aalto University School of Science, Aalto Print, Helsinki, Finland (11 2011)
17. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Incorporating clause learning in grid-based randomized SAT solving. Journal on Satisfiability Boolean Modeling and Computation 6(4), 223–244 (2009)
18. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search 107(2-3), 289–311 (2011)
19. Hyvärinen, A.E.J., Manthey, N.: Designing scalable parallel SAT solvers. In: Proc. SAT 2012. LNCS, vol. 7317, pp. 214–227. Springer (2012)
20. Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In: desJardins, M., Littman, M.L. (eds.) Proc. AAAI 2013. AAAI Press (2013)

21. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Proc. POPL 2014. pp. 607–618. ACM (2014)
22. Martins, R., Manquinho, V.M., Lynce, I.: An overview of parallel SAT solving. Constraints 17(3), 304–347 (2012)
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001. pp. 530–535. ACM (2001)
24. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337 – 340. Springer (2008)
25. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Proc. RTA 2005. LNCS, vol. 3467, pp. 453 – 468. Springer (2005)
26. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Proc. SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer (2006)
27. 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)
28. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proc. CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer (2013)
29. 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)
30. Sebastiani, R., Tomasi, S.: Optimization in SMT with LA(Q) cost functions. In: Proc. IJCAR 2012. LNCS, vol. 7364, pp. 484–498. Springer (2012)
31. 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)
32. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21(4), 543–560 (1996), `citeseer.ist.psu.edu/zhang96psato.html`