Parallel Hybridization for SAT

An Efficient Combination of Search Space Splitting and Portfolio

Rodrigue Konan Tchinda and Clémentin Tayou Djamegni

Department of Mathematics and Computer Science
Faculty of Sciences
University of Dschang
P.O. Box 67, Dschang-Cameroon
{rodriguekonanktr, dtayou}@gmail.com

ABSTRACT. Search space splitting and portfolio are the two main approaches used in parallel SAT solving. Each of them has its strengths but also, its weaknesses. Decomposition in search space splitting can help improve speedup on satisfiable instances while competition in portfolio increases robustness. Many parallel hybrid approaches have been proposed in the literature but most of them still cope with load balancing issues that are the cause of a non-negligible overhead. In this paper, we describe a new parallel hybridization scheme based on both search space splitting and portfolio that does not require the use of load balancing mechanisms (such as dynamic work stealing).

RÉSUMÉ. Les deux principales approches utilisées dans la résolution parallèle du problème de satisfiabilité propositionnelle sont DPR (Diviser Pour Régner) et portfolio. Chacune d'elles comporte des forces et des faiblesses. La décomposition dans DPR permet d'améliorer le speedup sur les instances satisfiables tandis que la compétition dans les portfolios accroit la robustesse. Plusieurs approches hybrides pour la résolution parallèle de SAT ont été présentées dans la littérature mais la plupart d'entre elles souffrent encore des problèmes dus aux mécanismes de rééquilibrage dynamique de charges qui sont à l'origine d'un surcoût non négligeable. Nous décrivons dans ce papier un nouveau schéma d'hybridation parallèle basé sur les deux approches DPR et portfolio ne nécessitant pas la mise en œuvre des mécanismes de rééquilibrage de charges (tels que le vol de tâche).

KEYWORDS : SAT, portfolio, search space splitting, parallel hybridization

MOTS-CLÉS : SAT, portfolio, DPR, hybridation parallèle

1. Introduction

The Boolean Satisfiability Problem (SAT) consists of determining whether there exists an assignment of truth values to variables of a given propositional logic formula in order to make it evaluate to true. This problem of great importance in Computer Science is a subject of special attention since the advent of modern SAT solvers based on the so-called CDCL (Conflict-Driven Clause Learning) procedure [17, 26, 28, 12, 11]. SAT is known to be NP-complete [9] and therefore is very hard to solve (unless $P = NP$). Despite this theoretical hardness, recent researches in the last two decades have resulted in very efficient SAT solvers that are able to solve industrial formulas with millions of variables and clauses in very little time. This great success has led to their use in many other fields including formal verification, planning, bioinformatics, cryptanalysis, etc. Faced with ever increasing need of performance and because of microprocessors' frequency limitation due to technological constraints, the efficiency of current state-of-the-art sequential SAT solvers is no longer sufficient since many industrial instances are still out of their reach.

Researchers then turned to efficient parallelization of SAT [13, 14, 19, 20] since the increase in the computing power of microprocessors today has resulted in an increase in their number of cores. Nowadays, there are two main approaches used for this purpose namely search space splitting and portfolio, each of which having its strengths and weaknesses.

Many hybrid approaches [8, 21, 19, 25, 18] have been proposed for parallel SAT solving but most of them still suffer from load balancing issues which are the cause of a non-negligible overhead. Our aim in this paper is to propose a new hybridization scheme that overcomes workload balancing issues while inheriting the best features of search space splitting and portfolio approaches.

The remainder of this paper is organized as follows: Section 2 briefly recalls some basic concepts and gives an overview of parallel SAT solving. Section 3 exposes our hybrid approach and in Section 4 we present our implementation followed by experimental results. We present some related work in Section 5 and finally conclude our paper in Section 6 while pointing out some future research directions.

2. Preliminaries

We assume that the reader is familiar with the Boolean Satisfiability problem; however, we recall here some basic concepts used in solving this problem. The interested reader may refer to [7, 6, 22] for more details.

2.1. Definitions and Notations

A *Boolean variable* is a variable that can be assigned only two possible values: *true* ($\top$ or 1) or *false* ($\bot$ or 0). A *literal* is either a Boolean variable (*positive literal*) or its negation (*negative literal*). A *clause* is a disjunction of literals (i.e. literals connected with $\vee$). Propositional formulas are commonly represented in *Conjunctive Normal Form (CNF)* i.e. as a conjunction of clauses (clauses connected with $\wedge$). A CNF formula can be seen as a set of clauses where each clause is a set of literals. An *interpretation* or *assignment* (or a *truth assignment*) is a map $\sigma : \mathcal{V} \rightarrow \{0, 1\}$ which associates a truth value to each variable of $\mathcal{V}$. If $\mathcal{V}$ is a subset of variables of a formula $\mathcal{F}$ then σ is called a *partial assignment* of $\mathcal{F}$. A truth assignment $\sigma : \mathcal{V} \to \{0,1\}$ can be represented as a set of literals $\mathcal{I}$ such that for every variable $x \in \mathcal{V}$, $x \in \mathcal{I}$ iff $\sigma(x) = 1$, $\neg x \in \mathcal{I}$ iff $\sigma(x) = 0$ and x is unassigned iff $\{\neg x, x\} \cap \mathcal{I} = \emptyset$. A literal l is satisfied under an assignment $\mathcal{I}$ if $l \in \mathcal{I}$ and falsified if $\neg l \in \mathcal{I}$ where $\neg l$ denotes the opposite literal of l i.e. the literal that evaluates to *true* when l is *false* and *false* when l is *true*. A clause is said to be *satisfied* under an assignment when it contains at least one satisfied literal, and is *falsified* if all its literals are falsified. An empty clause is a clause with no literals: it is always falsified. A clause is *unit* under a partial assignment when all its literals are falsified except one which is unassigned. A CNF formula is satisfied under an interpretation $\mathcal{I}$ if all its clauses are satisfied under $\mathcal{I}$: $\mathcal{I}$ is then called a *model* of $\mathcal{F}$. A CNF formula $\mathcal{F}$ is said to be *satisfiable* if there exists an assignment under which $\mathcal{F}$ is satisfied ; $\mathcal{F}$ is *unsatisfiable* otherwise. Given a CNF formula $\mathcal{F}$ and a literal l, we write $\mathcal{F}_{|l} = \{c | c \in \mathcal{F}, \{l, \neg l\} \cap c = \emptyset\} \cup \{c \backslash \{\neg l\} | c \in \mathcal{F}, \neg l \in c \text{ and } l \notin c\}$. $\mathcal{F}_{|l}$ denotes the simplified formula obtained from $\mathcal{F}$ by removing all clauses $c \in \mathcal{F}$ such that $l \in c$ and $\neg l$ from clauses containing it. This simplification can be extended to a set of literals $\{l_1, \cdots l_k\}$; thereby $\mathcal{F}_{|\{l_1,\cdots,l_k\}}$ is the formula obtained from $\mathcal{F}$ by successively applying the previous simplification rule on $l_1, l_2, \cdots$ and l_k i.e. $\mathcal{F}_{|\{l_1,\cdots,l_k\}} = (...(\mathcal{F}_{|l_1})_{|l_2}...)_{|l_k}$. *Unit propagation* is the application of the rule $\mathcal{F}_{|x}$ for each unit clause $\{x\} \in \mathcal{F}$ until a clause in $\mathcal{F}$ is falsified or $\mathcal{F}$ does not contain unit clauses anymore.

The Boolean Satisfiability Problem (SAT) consists of deciding whether a given propositional formula $\mathcal{F}$ is satisfiable or not; in other words, SAT consists of determining if there exists a truth assignment to variables of $\mathcal{F}$ which can make it evaluate to true.

2.2. Parallel SAT Solving

The two main approaches commonly used in parallel SAT solving are *search space splitting* and *portfolio*. Each of them has its strengths and its weaknesses. In this section, we aim to present those two approaches.

2.2.1. Search Space Splitting

In this approach, the search space is partitioned into several disjoint parts or branches which can be treated in parallel. The partition function used to split the search space takes as input a formula $\mathcal{F}$ and outputs a set $\mathcal{P} = \{\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_n\}$ of sub-formulas such that $\mathcal{F}$ is satisfiable if there exists a satisfiable $\mathcal{F}_i \in \mathcal{P}$ and is unsatisfiable if every $\mathcal{F}_i \in \mathcal{P}$ is unsatisfiable. Due to the difficulty of predicting the time needed to complete a specific branch of the search space [19], the partitioning is usually done dynamically rather than statically. Therefore, parallel solvers based on search space splitting dynamically partition the search space, assigning available work to the available threads during runtime. The splitting of the search space is usually done by means of the so-called guiding path. This concept of guiding path was initially introduced by [27] and has been much used in parallel SAT solvers. It describes the current state of the search process by recording the list of variables to which the solver gave a value up until the current point of execution. Guiding paths are used to distribute work among threads during the search process. Since some guiding paths can be easier to solve than others, a workload balancing strategy such as dynamic work stealing [16] is used to supply tasks to idle threads during execution. Thus, when a thread becomes idle, it can request new guiding paths from another thread or from the master thread depending on the chosen collaboration scheme. The search process is stopped when a model is found by one thread or when all guiding paths are solved. The main drawback here mentioned in [18] is the load balancing issue since it is hard to predict time needed to complete a specific branch of the search tree and therefore difficult to find a partition that balances work among threads. In addition, using dynamic load balancing in the context of SAT can bring further issues such as the *Ping-Pong phenomenon* [15] which occurs when division of the search space using a variable repeatedly provides two subspaces with one that is very easy to solve. Hence, workers spend a huge amount of time on splitting operations and communications instead of actually solving the problem itself. Another issue is *useless division* [2] where the resulting sub-formulas are identical. These issues are the source of an important overhead in parallel SAT solvers based on search space splitting. However, through the splitting of the search space, good speedup can be reached more frequently on satisfiable formulas.

2.2.2. Portfolio

The portfolio approach exploits the complementarity between different sequential CDCL strategies that compete and cooperate on the same formula. To be efficient, a good crafting of the solver is required in order to perform the search in the best possible way. Portfolio solvers generally run different incarnations (also referred to as threads or solvers) of the same sequential solver on the same instance: the rationale is the high sensibility to parameter tuning which constitutes the main weakness of modern solvers [13]. For instance, a small change of parameters related to the restart strategy, the learned clauses database cleaning strategy or the branching heuristic can lead to a solver with completely different performances. Threads of the portfolio then use different parameters tuning that lead to complementary strategies in order to cover the search space in the best possible way. In order to improve the performance of the system beyond the performances of its individual threads, information sharing has been introduced in portfolio solvers. This information includes learned clauses, variable activities, equivalent variables etc.

Portfolio has the advantage that it does not need load balancing and is simple to implement. However, a real challenge with portfolio approach is the difficulty to guarantee diversification of the search through algorithms that complement each other and therefore difficult to ensure scalability [19].

3. Our Parallel Hybridization Scheme

Decomposition in search space splitting is beneficial since it can help achieve better speedup while competition in portfolio with different search strategies can help explore the search space in different and complementary manners without the need of load balancing. It is then natural to think of a hybrid approach that can inherit those characteristics in order to perform better. We present in this section a new hybridization scheme for parallel SAT solving. The principle of our approach is described as follows:

We start with the decomposition of the search space into multiple disjoint parts as in search space splitting approach (Appendix Fig. 2a). This decomposition can be carried out by any partition function and also, can be performed in parallel for more efficiency. At this level, the better the partition function is, the better the resulting algorithm. After this partitioning, solvers of the portfolio are placed in a regular manner over the different parts (Appendix Fig. 2a): in this way, the chances to quickly discover a solution on satisfiable benchmarks are increased. Each solver of the portfolio has its own strategy and migrates (or jumps) through subspaces (also referred to as subproblems, sub-formulas, parts or guiding paths) in a round robin fashion, looking for a solution (Appendix Fig. 2b). This migration is performed even if the current guiding path is not yet solved and is directed by a heuristic that helps threads escape from subspaces that seem not interesting. The threads could however branch to this guiding path later and with all additional information gathered during the search elsewhere, this subspace could be interesting again. Notice that we use here the *interestingness* but not the *hardness* since a subspace can seem difficult to solve according to a particular thread but very easy to solve by another one with a different strategy or by the same thread at some point in the future. The difference is that when a subspace seems difficult to solve according to a single thread or a subset of threads, then it is *uninteresting*. But when it seems difficult according to every thread after several attempts then it is considered *hard*. At this level the interestingness of the subspace can be expressed according to the number of conflicts achieved within it, the average learned clause sizes or LBD scores [3, 4] in this subspace, the evolution of the search process or any other measure. Whenever a solver encounters an unsatisfiable sub-formula, it marks it to prevent other solvers from branching to it again (Appendix Fig. 2c). When one solver finds a solution (i.e. a model) or when all sub-formulas are unsatisfiable, the search process is stopped (Appendix Fig. 2f). If it happens at some point of execution that only a single subspace remains to be explored (Appendix Fig. 2d), the solving process is temporarily stopped in order to repartition the remaining subspace (Appendix Fig. 2e) which is considered as the hardest one among the initial parts. The rationale is that when a single unsolved subspace remains, it means that several threads with different and complementary strategies and sometimes with multiple attempts have tried to solve it without success. This subspace is then not only considered as uninteresting but declared as hard or difficult and is therefore split again. Relevant learned clauses are still exchanged as in classical parallel solvers in order to improve the efficiency of the system. In addition, threads can perform several restarts on the same sub-formula: this can be useful since it helps achieve the same objectives as the standard restart strategy in CDCL SAT solvers but within a specific subspace.

With this hybridization scheme, we can benefit from the strengths of both parallel approaches while eliminating some of their individual weaknesses. At first, there is no need to introduce a workload balancing mechanism as in search space splitting since at no time in the solving process, a solver becomes idle. Parallel solvers based on our scheme are likely to reach more often a super-linear speedup through the splitting of the search space, and the use of a portfolio with multiple search strategies helps explore subspaces with different and complementary methods which therefore increases robustness. Furthermore, the use of a migration heuristic helps threads escape from uninteresting subspaces and consequently directs them toward subspaces that are likely to be rapidly solved.

Useless splitting is no longer dramatic since even if it happens that threads work on identical sub-formulas, the various heuristics that they use help them explore it differently. The *ping pong phenomenon* is avoided here because each thread does not just work on a single part of the search space but instead, it works on the entire set of parts and no thread is stopped during the search in order to split its work. Furthermore, our approach is easier to implement compared to the search space splitting approach which requires dynamic work stealing.

Unlike classical portfolio, the diversification is well controlled through the splitting of the original search space into multiple disjoint subspaces. Thus, changing sub-formula can help improve diversification that is also enhanced by the use of many search heuristics and the initial placement of the threads over subspaces. As regards the intensification, a migration heuristic is used to control the amount of time a thread spends in a particular subspace.

In the literature, it is not clear how to characterize a hard subspace. The number of conflicts, the average LBD scores and the average backjumping levels are some measures commonly used to determine the hardness of a subspace. In the parallel context, these measures are sometimes taken according to a single thread. However, a subspace can seem difficult to a thread with its strategy while being very easy to solve by another thread with a different strategy. Moreover, even with a single thread, a subspace might seem difficult at the present moment but becomes very easy to solve later with additional information learned during the search performed elsewhere. In contrast, we consider a subspace hard when several threads having different search heuristics strengthened by information sharing have attempted to solve it without success. It is the case when during our proposed approach it remains a single sub-formula while the others have been proved unsatisfiable. Unlike some hybridization techniques which use a portfolio to solve difficult branches of the search space, once a sub-formula is found hard, then it is split again. The rationale is that when a task is difficult or large, it would be more natural to split it into small parts before solving rather than giving the whole task to each of the available workers.

4. Implementation Details and Experiments

We implemented our approach on top of the solver *PeneLoPe* [1] (the 2014 SAT Competition version), a parallel portfolio SAT solver which is in turn built on top of *Manysat* [13] and *Minisat* [10]. We gave to this modified version of *PeneLoPe* the name *PeneLoPe-DPRFolio*. *PeneLoPe* was chosen because of its good performance in previous SAT competitions and additionally because it is built on top of the famous SAT solvers *Minisat* and *Manysat* that are well documented and easy to modify. Note that we only empirically compared our solvers with the base solver on which they are built but not other parallel hybrid SAT solvers or parallel solvers based on search space splitting. The reason is that most of these solvers are not easily available online or they are implemented for special environments using non-standard middlewares. Nonetheless, in Section 5 we compared our approach to others based on how they work.

For the partitioning heuristic, we used a weak portfolio [18] to choose 3 partition variables. The principle of weak portfolio is to run a first stage of portfolio for a small amount of time usually expressed in term of number of conflicts. After that, variables that are the most active according to all threads of the portfolio are chosen to split the search tree. To do so, in each thread, variables are ranked in descending order according to their activities. Afterwards, each variable is given a score which corresponds to the sum of its ranks in each thread. Finally, the variables with the lowest score (which correspond to the most active ones) are chosen for partitioning. Weak portfolio has the advantage that easy formulas can be solved without any splitting of the search space.

We also used assumptions [10] to indicate the guiding path each thread must branch on: in this manner, learned clauses could be shared among threads without restriction and unsatisfiable instances could be sometimes solved by a single thread when a top-level (level 0) conflict has been found i.e. without proving the unsatisfiability of all the parts of the whole partition. It is worth mentioning that when it remains a single unsolved guiding path $G = \{l_1, \cdots l_k\}$, then its literals can be considered as units since the whole formula is satisfiable if and only if $\mathcal{F}_{|G}$ is satisfiable. To decide the moment at which a thread jumps from one part to another, we used the following heuristic based on LBD [3, 4, 5] which is one of the measures used to predict learned clause quality and that has shown very good performances in recent SAT competitions: every thread is forced to make at least 100 conflicts in a subspace it branches to before any jump; unless the corresponding sub-formula is earlier found unsatisfiable. This is used to prevent threads from jumping every time without performing a significant search in the subspaces they just branch to. Each thread jumps from its current sub-formula to the next one whenever the average LBD scores of all learned clauses generated since the branching on this subspace multiplied by a constant α ($0 < \alpha < 1$) called *jump factor* is greater than the global mean of the LBD scores computed since the launching of the thread. The rationale here is that if in one subspace, a solver is learning clauses with bad LBD scores, then it may not be an interesting subspace and therefore, jumping to another one can prevent it from getting stuck in it. More formally, if M is the current mean of the LBD scores since the entering in the current subspace and M_G is the mean since the launching of the thread, then this thread must jump to the next unsolved subspace if $M \times \alpha > M_G$. This heuristic is similar to the one used in LBD restart strategy [4, 5] but does not need the use of a bounded queue. According to the value of the jump factor α, we differentiated three versions of *PeneLoPe-DPRFolio*: *PeneLoPeDPRFolio-0.6*, *PeneLoPeDPRFolio-0.7* and *PeneLoPeDPRFolio-0.9* with respectively the *jump factor* set to 0.6, 0.7 and 0.9. Notice that a *jump factor* close to 1 indicates that the thread jumps more frequently.

All our experiments were conducted on the StarExec [1] [23] cluster infrastructure running Red Hat Enterprise Linux Server release 7.2 (Maipo). Each node of this infrastructure has two 4-core (2.4GHz) Intel processors, but we only had the possibility to use one of them. This means that we could only launch 4 threads in parallel and that is why all the solvers we used (*PeneLoPe* included) were tuned to use 4 threads. We also used deterministic mode to ensure reproducibility.

Experiments were carried out on the 100 parallel track benchmarks of the SAT-Race 2015 [2]. Notice that the 100 benchmarks (out of the 300 benchmarks used in the SAT-Race 2015) of the SAT-Race were selected by the organizers based on their hardness using a measure presented in the competition page. Solvers were used without any preprocessing step. Each instance was given a wall clock time limit of 1800 seconds and a memory limit of 24 GB.

Table 1 summarizes our results. In this table, the number of solved instances is indicated for both satisfiable and unsatisfiable benchmarks and the total run time used to solve these instances is specified in brackets.

Table 1 – Experiment results on the 100 hardest benchmarks of the parallel track of the SAT-Race 2015

Solvers	#SAT (time)	#UNSAT (time)	Total
PeneLoPe (SC 2014)	26 (17,639 s)	7 (5,447 s)	33
PeneLoPeDPRFolio-0.6	27 (11,826 s)	**7 (4,058 s)**	34
PeneLoPeDPRFolio-0.7	27 (12,161 s)	7 (4,221 s)	34
PeneLoPeDPRFolio-0.9	**32 (22,504 s)**	6 (3,393 s)	**38**

This table clearly indicates that our hybrid approach outperforms the original solver *PeneLoPe* especially on satisfiable instances where our solvers solve respectively 1 and 5 satisfiable instances more than *PeneLope*. Furthermore, we can notice that the main

1. https://www.starexec.org/
2. https://baldur.iti.kit.edu/sat-race-2015/

improvement is not the additional number of solved instances but the total run time used to solve these instances. We have for instance, *PeneLoPeDPRFolio-0.6* which solved 27 satisfiable instances in 11,826 seconds while *PeneLoPe* solved 26 instances in 17,639 seconds; so, the time used by *PeneLoPe* to solve 26 instances is far greater than the time needed by *PeneLoPeDPRFolio-0.6* to solve 27 instances. In Fig. 1, we have the cactus plots on the total solved benchmarks and on the total satisfiable solved instances. These plots clearly show that *PeneLoPeDPRFolio* outperforms *PeneLoPe* and also that the performance is mainly gained on satisfiable instances. This can be explained by the introduction of search space splitting in our approach which helps improve speedup on satisfiable instances.

These results lead us to the conclusion that our solvers perform well on hard satisfiable instances.

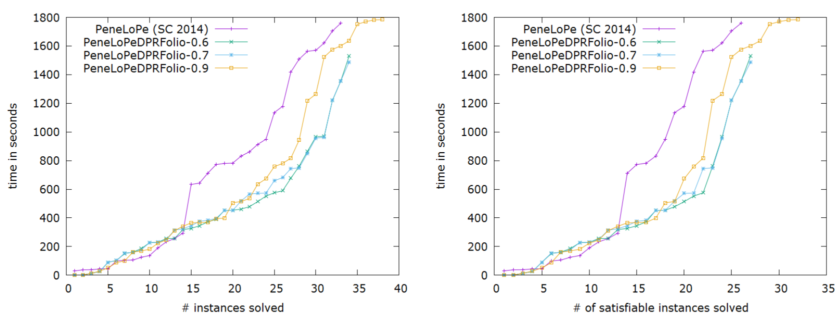


Figure 1 – Cactus plot on the 100 hardest benchmarks of the SAT-Race 2015: on the left, the cactus plot on all (satisfiable and unsatisfiable) instances. On the right, we have the cactus plot on satisfiable instances only

5. Related Work

Many hybrid approaches for parallel SAT solving have been proposed over the years.

BLOCHINGER [8] proposed to use an adaptive competition by starting with a search space splitting strategy and switching into a portfolio approach when a particular hard region of the search space is encountered. As we can see, this approach needs to balance workload between threads during the search space splitting phase; that is why the author used dynamic work stealing for that end. In addition, the hardness of a subspace is only determined by a single thread with a single configuration. In contrast to this latter remark, in our approach, the hardness of a subspace is determined by all the threads participating in the portfolio and is set as such when several threads with several configurations, strengthened by shared information have attempted to solve it without success.

OHMURA *et al.* [21] in their solver c-SAT tried to take advantage of a high number of machines by combining search space splitting with a portfolio approach. Here again, dynamic workload balancing is necessary to prevent idleness of workers. Furthermore, since a worker can only abandon a subspace when this latter is solved (satisfiable or unsatisfiable), then a worker can get stuck into a subspace that seems to be very difficult for it while there are some other subspaces that it can solve very quickly. To overcome this limitation, we allowed threads to temporarily leave one part in favour of another even if their current sub-formula is not yet solved. These threads could however come back later to the abandoned subspace and the additional information gathered during the search may help them to solve it more efficiently.

NISHANT *et al.* [24] proposed to use a search space splitting at the high level to divide the search space into multiple disjoint parts and assign each part to a portfolio of solvers. They do not use any kind of workload balancing in their methods; hence processes that are assigned easy guiding paths rapidly become idle.

MARTINS *et al.* [18] proposed to begin with an initial stage of search space splitting, switching to a portfolio approach when load balancing becomes an issue or when a cutoff is reached. After this switch, the solver does the remaining work in a portfolio mode. The motivation of the authors is to use search space splitting when this approach is more efficient and to change to a portfolio approach when difficulties arise. The transition between the two modes is heuristically done. Here once again the dynamic load balancing is necessary before the transition. In addition, the transition between search space splitting and portfolio is initiated according to the point of view of a single method since before the transition all the threads use the same heuristic.

6. Conclusion and Future Work

In this paper, we have presented a new parallel hybridization scheme for SAT. Our approach divides the search space into disjoint parts and then, places the solvers of the portfolio over these parts in a regular manner and lets them migrate from one part to another even if the current part is not yet solved. It uses heuristics for the choice of partition variables and the migration moment. Our approach does not need any workload balancing mechanism and can achieve good speedup on hard satisfiable instances. We integrated it in the solver *PeneLoPe* and performed some experiments and comparisons. The results showed that our hybridization scheme actually help improve the performance of the solver *PeneLoPe* especially on hard satisfiable instances.

Results suggest that the jump factor α can have a significant impact on the performance of the solver. So one further research direction is to investigate the real impact of this factor in the search process.

Bibliography

[1] Gilles Audemard, Benoit Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Penelope in SAT competition 2014. *SAT COMPETITION*, page 58, 2014.

[2] Gilles Audemard, Benoit Hoessen, Said Jabbour, and Cédric Piette. An effective distributed d&c approach for the satisfiability problem. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 183–187. IEEE, 2014.

[3] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and unsat. In *Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[5] Gilles Audemard and Laurent Simon. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[6] Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[8] Wolfgang Blochinger. Towards robustness in parallel SAT solving. In *PARCO*, pages 301–308, 2005.

[9] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[10] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[11] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000.

[12] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[13] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[14] Steffen Hölldobler, Norbert Manthey, Peter Steinke Van Hau Nguyen, and Julian Stecklina. Modern parallel SAT-solvers. *TR 2011–6*, 2011.

[15] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.

[16] Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

[17] JP Marques-Silva and KA Sakallah. Grasp-a new search algorithm for satisfiability. ICCAD, 1996.

[18] Ruben Martins, Vasco Manquinho, and Inês Lynce. Improving search space splitting for parallel SAT solving. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 336–343. IEEE, 2010.

[19] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[20] Tarek Menouer and Souheib Baarir. Parallel satisfiability solver based on hybrid partitioning method. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 54–60. IEEE, 2017.

[21] Kei Ohmura and Kazunori Ueda. c-SAT: A parallel SAT solver for clusters. In *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 524–537. Springer, 2009.

[22] Laurent Simon. Reasoning with propositional logic: From sat solvers to knowledge compilation. In *A Guided Tour of Artificial Intelligence Research*, pages 115–152. Springer, 2020.

[23] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: a cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, pages 367–373. Springer, 2014.

[24] Nishant Totla and Aditya Devarakonda. Massive parallelization of SAT solvers, 2013.

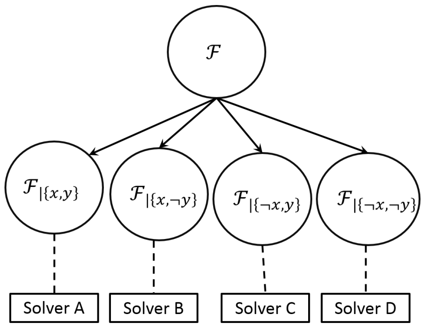
[25] Nishant Totla and Aditya Devarakonda. Massive parallelization of SAT solvers [cs262a project report], 2013.

[26] Hantao Zhang. Sato: An efficient prepositional prover. In *Automated Deduction–CADE-14*, pages 272–275. Springer, 1997.

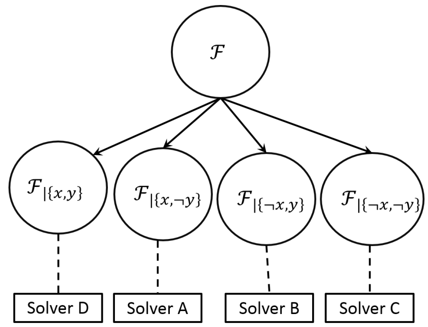
[27] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[28] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

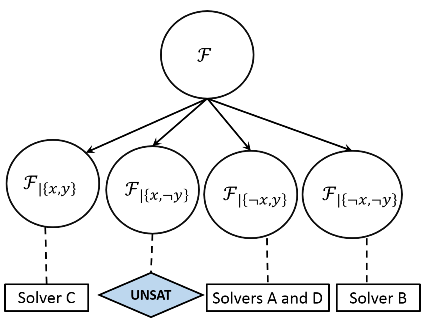
A. Appendix



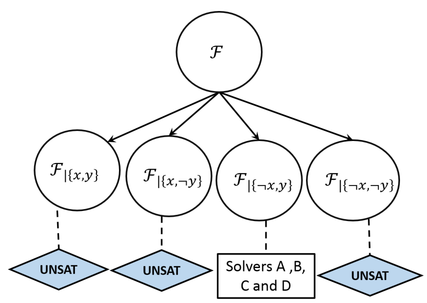
(a) Initially, solvers of the portfolio are regularly distributed over sub-formulas



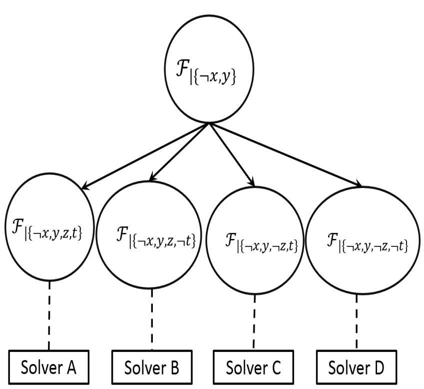
(b) Solvers migrate in a round robin fashion from one sub-formula to another



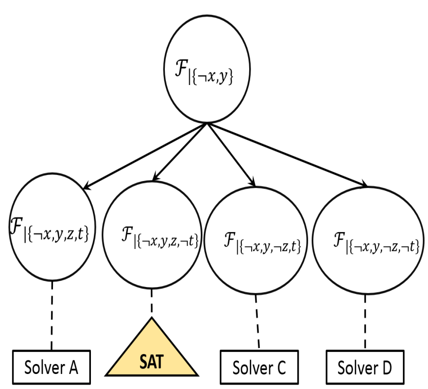
(c) Unsatisfiable sub-formulas are marked in order to prevent other solvers from branching to it again



(d) When it remains a single sub-formula, it is split again



(e) Partitioning



(f) The search is stopped if the solver finds a model or if all sub-formulas are unsatisfiable

Figure 2 – Schematic representation of the different steps of our approach