An adaptive parallel SAT solver

Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary

Univ. Lille-Nord de France. CRIL/CNRS UMR 8188, Lens, {name}@cril.fr

Abstract. We present and evaluate AMPHAROS, a new parallel SAT solver based on the divide and conquer paradigm. This solver, designed to work on a great number of cores, runs workers on sub-formulas restricted to cubes. In addition to classical clause sharing, it also exchange extra information associated to the cubes. Furthermore, we propose a new criterion to dynamically adapt both the amount of shared clauses and the number of cubes. Experiments show that, in general, AMPHAROS correctly adjusts its strategy to the nature of the problem. Thus, we show that our new parallel approach works well and opens a broad range of possibilities to boost parallel SAT solver performances.

1 Introduction

Papers dealing with SAT solvers usually begin by recalling the tremendous progress achieved on problems coming from industry. Recent results are indeed very impressive, and a large number of industrial problems, *e.g.* from planning [35], formal verification [11] and cryptography [40] are nowadays solved using a reduction to SAT instead of ad-hoc solvers. However, playing the devil's advocate, one can observe that progress has slowed down noticeably. It has become harder and harder to improve solvers dramatically. Furthermore, SAT suffers from its own success, since formulas to solve are more and more difficult.

At the same time, cloud computing is changing the landscape of computing science: it is now possible to request a virtually unlimited number of computing units that can be used within a few seconds. However, as it was pointed out during the last competition [37], parallel SAT solvers are not well scalable. Indeed, the winner of the parallel SAT track chose to only use half of the available cores. Thus, to benefit from the huge number of computing units, as in a cloud context, one must design new solvers architectures.

In the case of SAT solving, solvers can be divided into two categories. First and foremost, portfolio based approaches [1,8,13,23,24,36] run different strategies/heuristics concurrently, each on the whole formula. While computing the processes exchange information (generally in the form of learnt clauses) to help each other [1,7,23,24]. The second category of solvers uses the well known divided and conquer paradigm [2,15,16,25,26,39,41,42]. In such solvers, the search space is divided into sub-spaces, which are successively sent to SAT solvers running on different processors, so called workers. In general, each time a solver finishes its job (while the others are still working), a load balancing strategy is invoked, which dynamically transfers sub-spaces to this idle worker [15,16]. The sub-spaces can be defined using the guiding path concept [42], generated statically, *i.e.*, before the search [25,39], or dynamically, *i.e.*, during the search process [2,26,41]. As in portfolio solvers, learnt clauses can also be shared [18].

Even though the winners of the parallel track of the last SAT competitions are based on the portfolio paradigm, solvers based on the divide and conquer approach become increasingly more efficient (TREENGELING [12] a solver based on this paradigm was ranked second in the last competition). It is in this context that we propose AMPHAROS, a new parallel SAT solver, which follows the divide and conquer approach. Our long term objective is to develop a SAT solver for the cloud and this paper is a first step in this direction. In our approach, the formula is partitioned using cubes (as in [41]). One process, named MANAGER, is dedicated to managing these cubes. Then, solvers work on the formulas induced by those cubes. In constrast to other divide and conquer approaches, several solver may work on the same sub-problem and they can stop working before finding a solution or a contradiction. The latter is to avoid solvers being stuck on instances that turn out to be too hard for them. In that case, the solver asks the manager for another sub-problem. This sub-problem can either originate from an existing cube or from refining the current sub-problem. In our approach, the solvers select by themselves the dynamically generated cubes they try to solve. Additionally, two types of learnt clauses are shared: the classical shared clauses and others that are dependent on the cubes.

Since our goal is to solve SAT with a great number of computing units, it is important to propose a parallel architecture which adapts its strategy to the number of workers and the nature of the problem. To this end, we propose an approach which uses an adaptive algorithm that adjusts simultaneously and dynamically the number of clauses that are shared and the number of new cubes. This is possible thanks to a new measure that estimates if the search process has to be intensified or diversified. As we demonstrate in experiments, this measure works well and aligns with the stated goal. We show that when the search space needs to be diversified (resp. intensified), the proposed measure detects that the number of cubes must be increased (resp. decreased) and the number of shared clauses decreased (resp. increased).

2 Preliminaries

Due to lack of space, we assume the reader to be familiar with the essentials of propositional logic and SAT solving. Let us just recall some aspects of CDCL SAT solvers [32,30]. CDCL solving is a branching search process, where at each step a literal is selected for branching. Usually, the variable is picked w.r.t. the VSIDS heuristic [32] and its value is taken in a vector, called polarity vector, in which the previous value assigned to the variable is stored [34]. Afterwards, Boolean constraint propagation is performed. When a literal and its opposite are propagated, a conflict is reached, a clause is learnt from this conflict [30] and a backjump is executed. These operations are repeated until a solution is found or the empty clause is derived.

CDCL SAT solvers can be enhanced by considering restart strategies [20] and deletion policies for learnt clauses [3,6,19]. Among the measures proposed to identify the relevant clauses, the literal blocked distance measure (in short LBD) [6] is one of the most efficient. The clause's LBD corresponds to the number of different levels involved in a given learnt clause. Then, as experimentally shown by the authors of [6], clauses with smaller LBD should be considered more relevant.

It is well known that for several applications it is necessary to solve many similar instances [5,9,17]. To make solvers more effective in such a context, it is particularly useful to use assumptions to keep track of learnt clauses during the whole search. A set of assumptions is defined as a set of literals that are assumed to be true [17]. This set can be viewed as a cube, i.e. a cunjunction of literals (in the remainder of this paper, we denote cubes using square brackets, also we sometimes identify cubes with the formulas they imply), and the search is restricted to this cube. If during the search process, one needs to flip the assignment of one of these assumptions to false, the problem is unsatisfiable under the initial assumptions. In such a situation, it is possible to recursively traverse the implication graph to extract a clause that explains the reason of the conflict. Even if this problem seems close to the classical SAT problem, a special track of the last SAT competition has been dedicated to this issue [37] and several existing studies attempt to improve SAT solvers to deal with assumptions [4,28,33].

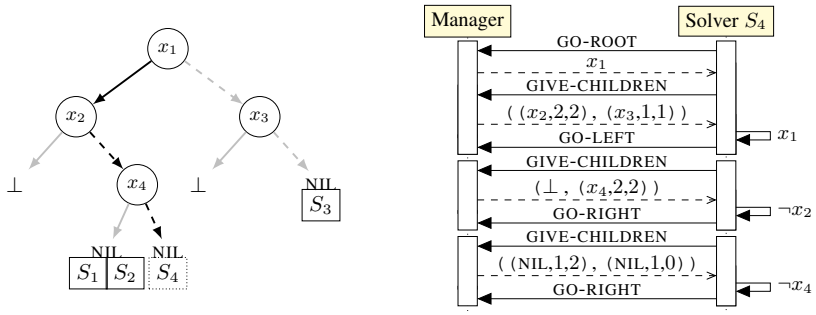
3 Tree management

The performance of divide and conquer approaches depends on both, the quality of the search space splitting, and how the sub-spaces are assigned to the solvers.

Even if AMPHAROS is a divide and conquer based solver, it is important to stress that, contrary to [38], it does not use the work stealing strategy. In our case, the division is done in a classical way as in [2,16]. More precisely, our approach generates guiding paths, restricted to cubes, that cover all the search space. This way, the outcome of the division is a tree where nodes are variables and the left (resp. right) edge corresponds to the assignment of the variable to true (resp. false). Then, solvers operate on leaves (represented by the symbol NIL) and solve (under assumptions) the initial formula restricted to a cube which corresponds to the path from the root to the related leaf. Fig. 1a shows an example of a tree containing three open leaves (cubes $[x_1,\neg x_2,x_4]$, $[x_1,\neg x_2,\neg x_4]$ and $[\neg x_1,\neg x_3]$), two closed branches (already proven unsatisfiable) and four solvers ($S_1 \dots S_4$) working on these leaves.

As we will see in Sect. 3.2, in our architecture, solvers can work on the same cube (as solvers S_1 and S_2 in Fig. 1a) and can stop working before finding a solution or a contradiction. In AMPHAROS, each time a solver shares information or asks to solve a new cube, it communicates with a dedicated worker, called MANAGER. Its main mission is to manage the cubes and the communication between the solvers (here CDCL solvers). Thus, when a solver decides to stop solving a given cube (without having solved the instance), it can ask the MANAGER to enlarge this one (see Sect. 3.3). Another situation where a solver stops, is once a branch is proved to be unsatisfiable. In this case, a message informs the MANAGER and the tree is updated in consequence (see section 3.4). In both cases, when a solver stops it goes through the tree and starts solving a new cube (potentially the same, see Sect. 3.2). The end of the solving process finally occurs either when a cube is proved to be satisfiable or when the tree is proved to be unsatisfiable.

This section describes the overall picture of our solver. First, in Sect. 3.1, the way the tree is initialized is presented. Then, the transmission and extension processes are respectively explained in Sect. 3.2 and Sect. 3.3. Finally a tree pruning rule is introduced in Sect. 3.4.



(a) Current tree (b) Sequence diagram

Fig. 1: Schematic overview of how the solver S_4 and the MANAGER interact to select a cube in the current tree represented Fig. 1a (a plain (resp. dotted) line means that the variable is assigned to true (resp. false)). On the sequence diagram, in Fig. 1b, we can see that seven messages are exchanged between the MANAGER and the solver before S_4 starts to solve the sub-problem induced by $[x_1, \neg x_2, \neg x_4]$. The path selectioned by S_4 is represented with black lines in the left picture.

3.1 Initialization

At the beginning of the search process, we initialize the workers. This step is required to setup the activity (related to VSIDS heuristic [32]), the polarity of variables and to create the root of the tree. To this end, all solvers try to solve the whole formula concurrently until a given amount of conflicts is reached (10,000 in our implementation). Note that this corresponds to solve an empty cube. In order to avoid performing the same search, the first descent of each solver (*i.e.* the choice of the variables and their polarity on the first branch) is randomized. Then, in the same manner as [31], the first solver reaching the maximum number of conflicts communicates its best variable with respect to the VSIDS heuristic to the MANAGER. This variable becomes the root of the tree. Consequently, the tree only contains two leaves, *i.e.* cubes are restricted to a single literal (a variable and its opposite). Regarding Fig. 1a, the selected variable was x_1 and the set of initial cubes was $\{[x_1], [\neg x_1]\}$.

3.2 Transmission

As already mentioned, a solver may stop the search before solving its instance. This situation occurs when it cannot solve the sub-problem associated to the cube with a number of conflicts less than a certain limit (10,000 in our implementation). The solver then contacts the MANAGER in order to select a potentially new cube to solve. The originality of our method is that a solver selects by itself one cube among all unsolved ones in the tree (corresponding to NIL leaves).

Fig. 1 shows a diagram sequence (Fig. 1b) that illustrates the exchanged messages when the solver S_4 requests a new cube from the MANAGER's tree (Fig. 1a).

A first message (GO-ROOT) is sent by the solver to ask for the root of the tree. It receives x_1. Then, at each step of the cube selection, the solver asks for the children of the previously received variable (with message GIVE-CHILDREN). The answer is composed of two triplets: one for each polarity of the current node. Each triplet is composed of

the child variable, the number of available leaves (NIL nodes) and the number of solvers working on these leaves, in that order. Considering Fig. 1b, the first message returns the triplet $(x_2,2,2)$ for the positive polarity of x_1 (the left branch contains two leaves and two solvers (S_1 and S_2)) and $(x_3,1,1)$ for the negative one.

The solver decides to go down either on the left (assigning positively the current variable) or on the right (assigning negatively the current variable) branch according to the values returned in these triplets. By default, it selects the branch where the number of working solvers is lower than the number of leaves. The idea is to cover the most of cubes and to dispatch solvers all over the tree. If this condition is true or false for both branches, the solver selects the branch according to its polarity vector [34]. Note that in this implementation, we do not know if some cubes do not contain solvers. After selecting its branch, the solver informs the MANAGER (with messages GO-LEFT or GO-RIGHT) and assigns the related literals using assumptions.

Thus, in our example of Fig. 1, the solver S_4 assigns x_1 (the root) positively using its polarity (as the condition previously mentioned is false for both branches). Since the branch related to x_2 is already proven unsatisfiable, S_4 does not have other alternatives to setting the literal x_2 to false. Finally, it has to set x_4 to false since the previous condition holds. Arriving at a leaf, the solver starts to solve the cube $[x_1,\neg x_2,\neg, x_4]$.

3.3 Extension

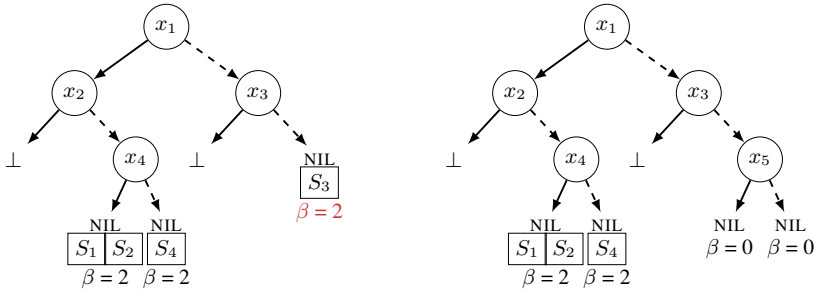
Initially, the tree contains only one variable and then two cubes to solve (see Sect. 3.1). To divide the original formula into a substantial number of cubes, we propose to dynamically extend the tree during the search. Recall that we do not use the work stealing strategy.

One associates to each leaf an integer variable β representing the presumed difficulty of a subproblem (cube). Each time a solver cancels its search on a given cube (associated with a leaf of the tree), the variable β of this leaf is incremented. Then, a large value of β expresses that a cube is potentially hard to solve. Note that a solver can increase several times the same variable β. When a solver stops its search and requests a new cube, the MANAGER increments the value β associated to the leaf on which the solver was working. When the β value of a leaf is greater or equal than the number of open leaves (*i.e.* NIL leaves) times an extension factor f_e then the tree is expanded on the given leaf. [1]

The extension is done in the following way. The last solver increasing the variable β returns its best boolean variable w.r.t. to VSIDS heuristic and two new leaves are created, extending the related cube. The β values of the two leaves are initialized to 0. Taking into account the number of open leaves, the more unsolved cubes the tree contains, the less extensions are performed. In this way and contrary to Cube And Conquer [41], our approach does not create too many cubes, regardless of the number of cubes already proven unsatisfiable. Since a leaf can contain many solvers, note that after extension, some solvers can work on a node that is not a leaf.

Fig. 2 shows an example of an extension. The tree (the same as in Fig. 1) contains 3 open leaves and some solvers work on these leaves. When, solver S_3 stop working on

[1] We will discuss about the definition of the extension factor in section 5.2.



(a) Extension requested by the solver S_3

(b) Extension accepted by the MANAGER

Fig. 2: The left picture represents the tree before the S_3's extension request was accepted. Since the value of β associated to the node satisfied the extension criterion, the MANAGER accepts this extension and modified the left tree to obtain the right one.

cube $[\neg x_1, \neg x_3]$ the associated β (in red) is incremented and becomes 3. The condition allowing an extension holds (we suppose that f_e is equal to one) and thus extension is performed. Solver S_3 that is responsible for the extension provides its best variable (x_5) to the MANAGER and the cube $[\neg x_1, \neg x_3]$ is expanded with the variable x_5 generating two new cubes. Note that the (red) β value initiator of this extension becomes useless since its associated node is not a leaf anymore. Solver S_3 is now free to ask the MANAGER a new cube to solve (see Sect. 3.2). Furthermore, in the next step, no matter which solver ask for extension, it will not be performed since the number of leaves is now equal to 4 (we suppose here that f_e remains unchanged and is still equal to 1).

3.4 Pruning

Because each sub-problem is solved under assumptions, when a cube is proved to be unsatisfiable, the solver (from which unsatisfiability is proved) computes a conflict clause (which is the negation of a subset of the literal assumptions). This information is transfered to the MANAGER which is able to compute a cutoff level in the tree search. The tree is simplified in consequence. Let us remark that a solver can directly prove the global unsatisfiability of the problem when the computed conflict clause is empty.

Moreover, if both children of a node are unsatisfiable then this node also becomes unsatisfiable. In that case, the node can be safely removed and the unsatisfiability is directly associated to the edge of its parent. Of course, this process is recursively applied until each node has at least one non-unsatisfiable child.

4 Clause exchange

In this section, we discuss the two ways of exchanging information in our solver AMPHAROS. We first explain how the clauses learnt by a solver are shared with the others and then we present an original approach to sharing local unit literals by taking advantage of our tree.

4.1 Classical Clause Sharing

It is well known that clause sharing noticeably improves the performance of parallel SAT solvers [24]. In our framework, solvers also share learnt clauses. However, contrary to the classical behavior where the clauses are directly shared between workers, for us information passes through the MANAGER.

Clause sharing from the solver side Once a solver reaches a threshold of conflicts (500 in our implementation), it communicates with the MANAGER to send and/or receive a set of clauses. Clauses to be sent are saved in a buffer which is cleared after each communication with the MANAGER. Good clauses with respect to initial LBD (less or equal to 2) are directly added to the buffer. Other clauses are also added, as in [7], if they participate in the conflict analysis. However, because we cannot share as many clauses as SYRUP, only clauses which obtain a dynamic LBD less or equals to 2 before being used twice in the conflict analysis procedure are shared.

In order to deal with imported clauses, solvers manage three buffers: `standby`, `purgatory` and `learnt`. Received clauses are stored in `standby`. In this buffer, clauses are not attached to the solver [3]. Every $4{,}000$ conflicts, clauses are reviewed: they can be transfered from a buffer to another, or be definitively deleted or kept in the current buffer.

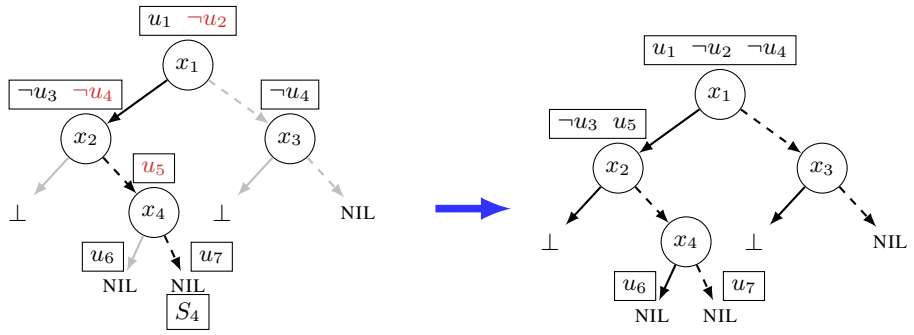
A clause from the `standby` buffer can be transfered to the `purgatory` buffer. Contrary to the `standby` buffer, clauses in the `purgatory` are attached to the solver and then participate to the unit propagation process. We discuss the criterion allowing a clause to be moved from `standby` to `purgatory`in Sect. 5.2.

In the same manner, a clause from the `purgatory` can be transfered to the third buffer `learnt` when it is used at least once in the conflict analysis process. The temporary buffer `purgatory` is used to limit the impact of new clauses on the learnt strategy reduction.

The reduction strategy used to clean these two additional buffers depends on a counter associated with each clause. The counter is incremented each time the associated clause remains in the same buffer. If the counter reaches a threshold (14 in our implementation), the clause is deleted. Note that the counter is reset each time a clause is moved from one buffer to another.

Clause sharing from the MANAGER side MANAGER collects learnt clauses from every solver and manages them. Learnt clauses are stored in a queue and the MANAGER periodically checks if they are subsumed or not. In practice, a single core is dedicated to the MANAGER. Thus, processing all clauses in the queue at once can be time consuming and can block communications between MANAGER and solvers. To avoid this situation, MANAGER checks subsumption by batches of $1{,}000$ clauses each.

MANAGER stores the learnt clauses that are not subsumed in a database and sends them each time a solver requests them. Of course, sent clauses are those that have not been already sent to the solver and that are not coming from it.



(a) S_4 gives $\neg u_2$ $\neg u_4$ u_5 to MANAGER (b) Unit literals management

Fig. 3: The left picture represents a decorated tree with the additional literals (in red) given by S_4. These additional literals are pull up, using the unsat (pull up u_5) and identical literals (pull up u_4) rules, to obtain the right tree.

4.2 Assumptive Unit Literals

A second way of exchanging information in our approach is transferring unit literals (which are propagated under some assumptive literals) between solvers and the MANAGER. In the following, we present where these literals originate from and how they are exchanged and managed.

Assumptive Unit Literals from the solver side Let us first recall that each solver works under an assumption A (this assumption can be empty) representing the cube to solve. When a literal $\ell \notin A$ is propagated thanks to a sub-assumption $A' \subseteq A$, this information can be spread to the MANAGER in order to be broadcasted to other solvers. More precisely, the solver communicates to the MANAGER that ℓ can be propagated with A'. From the other side, when a solver selects a branch (ie a literal ℓ') during cube transmission, it also receives the set of unit literals associated with ℓ' that can be propagated. Thus, the transmission of a cube (see Sect. 3.2) contains these additional messages. Hence, MANAGER takes care of a decorated tree containing guiding paths and the set of unit literals that have to be propagated at each branch. Figure 3.a shows an example of such a tree. When solver S_4 asks for a branch, it starts by recovering the set of unit literals $\{u_1\}$. It also propagates $\neg u_2$ (in red) giving this information to the MANAGER. It selects the branch x_1 and then, retrieves the literal $\neg u_3$ to propagate. in the same way, it also propagates $\neg u_4$, providing such assumptive unit literal to the MANAGER and so on.

As we will see in the experiments later, assumptive literals are very important. They are special clauses that clearly reduce the search space of a given branch. Consequently, the fact that a literal ℓ can be propagated from A' is taken into account in the solver by adding, in a dedicated database (this database is different from the aforementioned `learnt` buffer and is never cleaned up), a clause built with the negation of A' and the literal ℓ'. Remark that when $A' = \emptyset$ the literal ℓ' is unit and is added to the unit literals of the solver.

Assumptive Unit Literals from the MANAGER side When the MANAGER learns that a literal can be propagated from a subset of literals coming from an assumption, this information is communicated during cube's transmission and can be added in the last branch of the node associated with this sub-assumption. From this decorated tree, one can pull up unit literals from a branch to higher branches. This situation occurs either when a branch is proved unsatisfiable or when both branches of a node contain the same literal [29] (as highlighted Fig. 3). In the first case, all the literals of the non-unsatisfiable branch are pulled up to the father branch (as literal u_5). In the second case, only literals occurring in both branches are transmitted to the father branch (this is the case for literal $\neg u_4$). This process loops recursively until a fix point is reached. Remark that when no father branch exists (occurring when literals are moved from branches of the root node) then these literals are proved unit.

5 The Intensifcation/Diversification Dilemma

When several solvers run concurrently on a problem, they can perform redundant work. Identifying such a situation, it would be beneficial to modify the solvers' strategies in order to diversify the search. Nevertheless, due to clause sharing between solvers, exploring too different search spaces is also a handicap. Thus, in some situation focusing several solvers on the same part is required (intensification).

This paradigm, called intensification/diversification dilemma, has already been studied in the context of portfolio-based parallel SAT solvers. This issue can be addressed either statically, by using several solvers with orthogonal strategies [1,24,36], or dynamically, by modifying the solvers' strategies during the search. However, deciding when a solver must intensify or diversify its search is not easy and only few publications tried to deal with this problem [21,22]. Thus, in [21], a master/slave architecture is proposed, where masters try to solve the original problem (ensuring diversification), whereas slaves intensify their master's strategy. In [22], a measure to estimate the degree of redundancy between two solvers is presented. It considers that two solvers are closed when they have approximately the same polarity vector. The diversification process consists in modifying the way the phase of the next decisions is realized.

To the best of our knowledge, no criterion has been established to identify that several solvers execute redundant work except the measure based on the polarity mentioned before [22]. Unfortunately, this criterion is not applicable with many solvers (this measure has been initially proposed for a portfolio of four solvers). That is why a more scalable criterion is required.

5.1 Evaluating the Degree of Redundancy

We propose to measure the degree of redundancy by taking into account how many clauses that are shared between solvers are redundant. We use a list to store from the beginning the number of received clauses (st_r) and a second to store the number of kept clauses (st_k). Kept clauses are those which have not been removed during the subsumption process. Each time a solver comes back to the MANAGER (every $1{,}000$ conflicts in our implementation), it shares its clauses. The number of received (resp. kept) clauses since the beginning is pushed to st_r (resp. st_k) by the MANAGER.

The *redundancy shared clauses measure*, in short $rscm$, is defined for a step t w.r.t. a sliding window of size m (20,000 in our experiments) as the ratio between the number of clauses received during the last $t - m$ updates of st_r and the number of clauses kept during the same time. More precisely, we have $\forall j < 0$, $st_r[j] = st_k[j] = 0$:

$$\begin{aligned} rscm_t &= \frac{st_r[t] - st_r[t-m]}{st_k[t] - st_k[t-m]}, \text{ if } st_k[t] - st_k[t-m] \neq 0 \\ rscm_t &= st_r[t] - st_r[t-m], \text{ otherwise} \end{aligned} \tag{1}$$

First, note that when several solvers work on the same part of the search space, there is a high likelihood that learnt clauses by the different solvers are redundant. This means that the number of subsumed clauses is important and therefore the $rscm$ value is high. Conversely, when solvers are sparsed in the search space, there is a high probability that shared clauses are not redundant and then the $rscm$ value tends to be low. Consequently, a small value of the $rscm$ indicates that the solver needs to intensify the search, whereas a high value signifies that the solvers have to diversify their search space.

5.2 Intensification/Diversification Mechanisms based on the $rscm$ measure

It is possible to control in several ways how solvers explore the search space (shared clauses, solvers' heuristics, ...). In AMPHAROS, we choose to solve the intensification/diversification dilemma by controlling two criteria: the way the tree is extended (see Sect. 3.3) and the number of clauses which are transferred from the `standby` to the `purgatory` buffers (see Sect. 4.1). Thus, for us, diversifying (resp. intensifying) the search consists in increasing (resp. decreasing) these two parameters. Before introducing them, let us a summarize:

Few subsumed Clauses ($rscm$ is low)	**Many subsumed clauses** ($rscm$ is high)
Reduce extension	Favour extension
Increase the number of imported clauses	Limit the number of imported clauses
Intensification	**Diversification**

Extension guiding by the $rscm$ First, let us remark that each path from the root to a leaf represents a unique set of literals that splits the search space in a deterministic way. Thus, the bigger the tree, the higher the probability to run two solvers in totally different sub-problems. To control the tree grows, we define the extension factor fe introduced in the section 3.3 in the following way:

$$fe_t = \frac{1{,}000}{{rscm_t}^3} \tag{2}$$

Let us recall that this extension factor is used to define the threshold of misses that a cube can encounter before an extension is accepted. Hence, the smaller (resp. bigger) the $rscm_t$ value is, the bigger (resp. smaller) the fe value is and then the slower (faster) the tree extension is. Note that the cubic factor allows to decrease fe rapidly while fe is bounded (by 1,000) since when fe is high solvers run in concurrently and the tree is never updated. To prevent the tree from growing too quickly, we also bound the minimum value that fe can take by 10.

Condition to move from the `standby` to the `purgatory` When a clause is received by a solver it is possible that it is subsumed by a clause already present. This becomes highly probable when almost all shared clauses are found to be subsumed during the clause subsumption process. Thus, it seems natural that the number of accepted clauses (ie. the number of clauses transferred from the `standby` to the `purgatory`) increases (resp. decreases) when the $rscm$ value decreases (resp. increases).

As already mentioned (Sect. 4.1), in AMPHAROS the clauses freshly received are not directly attached to the solver. Thus, it is important to choose a clause selection criterion independant from the activation of clauses. To control the amount of clauses moved from the `standby` to the `purgatory` we use the notion of psm introduced in [3] and already used in the portfolio based SAT solver PENELOPE [1]. Recall that the psm of a clause represents the number of literals which are assigned to true by the polarity vector. Thus, a clause can be scored even if it is not used by the solver.

Then, in order to increase/decrease the number of clauses attached (and then transferred) in the `purgatory`, a criterion based on both the psm and $rscm$ values is proposed. This criterion is motivated by the observation from [3] that the clauses with a small psm value have a great potential to enter in conflict or be used during the search. Thus, a clause will be authorized to move from the `standby` buffer to the `purgatory` buffer when its psm value is less or equals than $\lfloor \frac{psm_{max}}{rscm_t} \rfloor$, where psm_{max} corresponds to the psm maximum limit accepted (set to 6 in our experiments). Consequently, clauses with a psm value of zero will be systematically accepted whatever the value of $rscm$. Whereas, clauses with a high psm value will be accepted if and only if they are probably not subsumed.

6 Experiments

We now evaluate AMPHAROS on the 100 benchmarks from the SAT-RACE 2015, parallel track [37]. During the last competition 53 (resp. 33) instances have been proved satisfiable (resp. unsatisfiable) by at least one solver and 14 instances remained unsolved. All experimentations have been conducted on 2 Dell R910 with 4 Intel Xeon X7550. Each node has 32 cores, a gigabit ethernet controller and 256GB of RAM. Time limit was set to 1,200 seconds per test (wall clock time). Then, for experiments executed with 64 cores, we use two different computers. All log files and additional pictures are available in http://www.cril.univ-artois.fr/ampharos/.

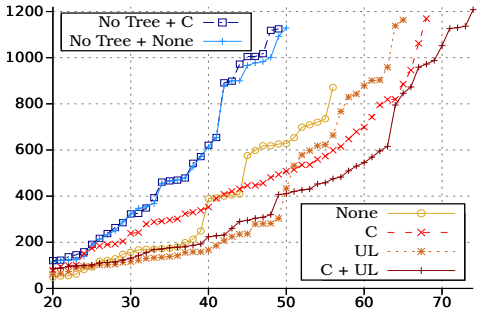
6.1 Communication management

Since in AMPHAROS a lot of messages have to be exchanged between the MANAGER and solvers, the management of the communications has to be very effective. Thus, we have opted for the open source Message Passing Interface implementation (Open MPI) to manage the communication on a low level.

The bottleneck imposed by the fact that the MANAGER has to all at once compute the subsumed clauses and communicate with the solvers, was a major problem. To avoid that solvers wait too long without work, a round robin architecture with non-blocking listening of solvers was put in place. Moreover, because the subsumption process can be time consuming, the clauses received to be checked are not treated at once (in our

Tree	Exchanges	SAT	UNS	Total
Yes	C + UL	**49**	**25**	**74**
Yes	C	47	21	68
Yes	UL	47	18	65
Yes	None	41	15	56
No	C	43	6	49
No	None	44	6	50

(a) Overview table



(b) Number of instances solved w.r.t. the time

Fig. 4: Comparing several versions of AMPHAROS on 64 cores. Tab. 4a gives the results of each version in term of solved instances. The columns represent, in that order, the fact that the tree decomposition is activated (**yes/no**), the kind of information exchanged (clauses (**C**) or/and unit literal (**UL**)), the number of SAT/UNSAT instances solved. Fig. 4b shows the number of solved instances (x-axis) w.r.t. the time (y-axis).

implementation packet of $1{,}000$ clauses are considered). Thus, the MANAGER communicates with a solver, then checks a set of clauses, and so on, until the time limit is reached or the problem is proved satisfiable/unsatisfiable.

6.2 Setup

AMPHAROS is a modular framework that allows to add easily new types of solvers. For these experiments three sequential SAT solvers have been used: GLUCOSE [6], MINISAT [17] and MINISATPSM [3]. Only a couple small changes have been implemented in these solvers. In order to manage the interactions with the MANAGER, all solvers implement a C++ interface. This interface grouped communication routines and methods used to deal with `standby` and `purgatory` buffers. The core of solvers has also been modified in order to avoid resetting everything at each call to SAT solver (restart, learnt deletion policies, ...). Moreover, as for the version of GLUCOSE presented in [4], when a solver restarts it does not go to decision level 0 but to the level of the last assumption. The clauses moved from the `purgatory` to the `learnt` buffer are simply incorporated into the learnt clauses database as if they were learnt by solvers themselves.

6.3 Results

The experimental evaluation is divided into four parts. First, we evaluate the different ingredients of AMPHAROS. Then, we study the scalability of our solver. Finally, we compare AMPHAROS to the state-of-the-art and study the impact of the $rscm$ measure.

On the impact of each component The benefit of the three optional components (tree decomposition (**Tree**), clauses (**C**) and unit literals exchange (**UL**)) of AMPHAROS has been studied experimentally. To this end, several versions of AMPHAROS have been executed on 64 cores. These experiments, reported in Fig. 4, show gradual improvements

when each of these options was taken into account in a cumulative way. From the table (4a) and the cactus plot (4b) several observations can be made.

First, Fig. 4a shows that whatever the combination of options is used, AMPHAROS is more efficient when the tree decomposition is used (Tree sets to true in the first column). The versions working on the initial problem in a competitive way, which could be regarded as portfolio parallel SAT solvers (with (C) or without (none) clause sharing), solve systematically less instances than the others running on sub-problem obtained from cubes. This shows the importance of how AMPHAROS solves the intensification/diversification dilemma using a tree decomposition.
Second, results show the importance of exchanging information between solvers. AMPHAROS that does not exchange information systematically solved less problems than the others which share clauses or unit literals. When we separately compare the two exchange options (C and UL), we observe that sharing clauses allows (as expected) to improve the solver on unsatisfiable problems. However, as pointed out in Fig. 4b, activating this option makes the solver slower on easy problems (solved with less than 600 seconds). This can be explained by the fact that the communication engendered to share clauses and manage them is significant and slows down the solvers on 'easy' problems.

Finally, as highlighted by these experiments, there is a synergy between the exchange options. Even if clause sharing drastically reduces the solver performance on easy benchmarks, combining this component with the unit literals allows one to deliver the most significant improvement in terms of number of successfully solved instances. From now, AMPHAROS is reported as the version using all components.

Scalability evaluation To evaluate the scalability of AMPHAROS we run it on 8, 16, 32 and 64 cores. Fig. 5 gives the number of solved instances w.r.t. the time by the different versions of AMPHAROS. It clearly demonstrates that our approach is highly scalable. The version running on 64 cores solves 49 SAT and 25 UNSAT benchmarks, that is 15%, 45% and 70% more benchmarks than the one running on 32 (44 SAT and 20 UNSAT), 16 (36 SAT and 15 UNSAT) and 8 (33 SAT and 11 UNSAT) cores, respectively. In order to show the efficiency of our approach with more computers linked over the network, we also ran it with 64 cores using 8 computers with 8 cores each (see the curve 8×8 on Fig. 5.). This version solves 46 SAT and 24 UNSAT benchmarks. Since we restrict the number of messages, we obtain similar results. Differences can be explain by the indeterminism of our approach.

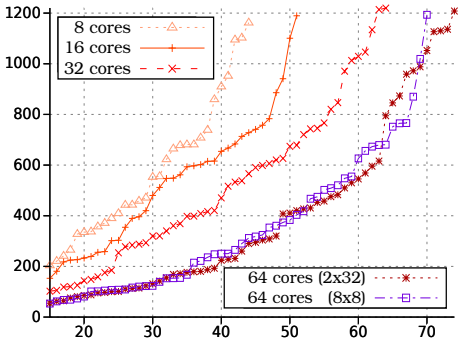


Fig. 5: Number of instances solved

AMPHAROS versus the state-of-the-art In order to evaluate AMPHAROS with respect to existing work, we choose to compare our approach with the three best solvers of the last SAT competition that ran in the parallel track. These solvers are (in their

rank order): SYRUP [7], TREENGELING and PLINGELING [12]. Because they do not run with MPI, and we have no processor with 64 cores, we execute them on 32 cores. We also compare our solver on 64 cores with the work stealing parallel SAT solver DOLIUS [2] and the portfolio solver HORDESAT [8]. Fig. 6 reports results obtained by these solvers.

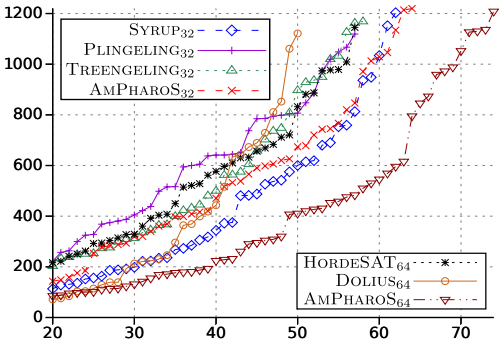
Let us first consider the experiments launched on 32 cores. As reported in Tab. 6a, AMPHAROS solves more instances than the other solvers. It is the best solver on the satisfiable benchmarks and solves the same number of unsatisfiable problems as TREENGELING (which is also a divide and conquer based method). Comparing to SYRUP and PLINGELING, we can see that our solver significantly outperforms both on satisfiable problems but it is less efficient on unsatisfiable ones. This can be partially explained by the fact that AMPHAROS essentially solves the unsatisfiable problems by totally refuting the tree (*i.e.* by closing all branches). Consequently, it seems that on 32 cores we do not have enough workers to achieve this goal within the time limit.

When considering the running time of the solvers, reported in Fig. 6b, we can observe that AMPHAROS is faster than TREENGELING and PLINGELING but it is slower than SYRUP. This can be explained by the fact that SYRUP solves several unsatisfiable problems in short time (the `6s` family for instance).

If we consider the experiments run on 64 cores, we can see that our approach is highly competitive. AMPHAROS is significantly better than DOLIUS and HORDESAT. Moreover, it is important to notice that during the competition SYRUP (the winner of the parallel track) only used 32 cores instead of the 64 cores available. Consequently, it is possible to conclude that AMPHAROS is more efficient than both TREENGELING and PLINGELING on 64 cores. More importantly, as reported in Fig. 6b, AMPHAROS is very effective since it solves more instances and faster.

Solver	**#thr.**	**SAT**	**UNS**	**Total**
AMPHAROS	32	**44**	20	**64**
SYRUP	32	36	**26**	62
TREENGELING	32	38	20	58
PLINGELING	32	31	**26**	57
AMPHAROS	64	**49**	**25**	**74**
HORDESAT	64	33	24	57
DOLIUS	64	33	17	50

(a) Overview table



(b) Number of solved instances w.r.t. the time

Fig. 6: Comparing AMPHAROS versus the state-of-the-art parallel SAT solver. Tab. 6a gives results of each solver in term of solved instances w.r.t. the number of threads (**#thr.**). Fig. 6b shows the number of solved instances (x-axis) w.r.t. the time (y-axis).

Let us stress that none of these solvers are deterministic. To be fair, we ran all solvers just once and report the obtained results (as it was done in SAT competition 2015).

Benchmarks Information		Time w.r.t. rscm					rscm statistics			
name	**sol.**	**1**	**3**	**5**	**10**	**D**	**min**	**max**	**avg**	**med**
hitag2-10-60-0-65	UNS	563	173	120	127	304	1.20	2.36	2.11	2.28
jgiraldezlevy.109	UNS	544	243	214	154	260	1.60	4.32	3.76	4.12
minandmaxor128	UNS	788	IN	IN	IN	972	1.09	1.37	1.25	1.23
jgiraldezlevy.33	SAT	776	339	386	169	288	1.63	4.58	3.32	3.82
56bits-12.dimacs	SAT	114	168	180	395	101	1.25	1.60	1.43	1.46
004-80-8	SAT	248	412	16	264	110	1.41	4.64	3.10	3.09

Table 1: This table presents the obtained results on a representative set of benchmarks. Each line corresponds to an instance, with its satisfiability, identified by the leftmost column. The next four columns give the WC time (reported in seconds) to solve the instance w.r.t. the value of $rscm$ (static (set to 1, 3, 5 and 10) or dynamic (D)). The rightmost reports stastictics on the value obtained by the dynamic computation of $rscm$.

Impact of the $rscm$ value To conclude this section, we evaluate the impact of the $rscm$ value on our solver's performance. To this end, we selected a representative set of benchmarks and ran four versions of AMPHAROS with different values of $rscm$ (1, 3, 5 and 10) and compared them with the dynamically chosen value. Let us recall that $rscm$ has an impact on both the tree extension and the amount of exchanged clauses. Moreover, recall that, as mentioned in Sect. 5.2, the extending factor fe is fixed to 10 when $rscm > \sqrt[3]{100}$. Thus, the difference between $rscm = 5$ and $rscm = 10$ is only the amount of shared clauses. Tab. 1 shows that these instances do not have the same comportment with respect to the $rscm$ value. Some problems need to extend more (`jgiraldezlevy`) and others need to extend less and exchange more (`minandmaxor128`). It is also important to note some benchmarks are unpredictable (`004-80-8`). As regards the dynamic adjustment, we observe that it is in average often close to the best value. The performance differences often come from the fact that the solver needs time to find the right $rscm$ value.

7 Conclusion

We proposed a new parallel SAT solver, designed to work on many cores, based on the divide and conquer paradigm. Our solver allows two kinds of clause sharing, the classical one and one more linked to the division of the initial formula. Furthermore, we proposed to measure the degree of redundancy of the search by counting the number of subsumed shared clauses. With this measure, we are able to adjust dynamically the search, resulting in a new way of controlling the dilemma of intensification/diversification of the search. Experiments show promising results.

Our main objective is to deploy a SAT solver among the cloud. Thus, this paper is a first step towards this goal and leads to many perspectives. We plan to run our solver on a cloud architecture using grid computing. For that, we plan to run several MANAGERs in parallel letting solvers go from a MANAGER to another one. For that, we need to choose more carefully variables used for the division. Many possibilities arise like the notion of blocked literals recently used for SAT solving [27,14]. Finally, we also need to improve performances on unsatisfiable instances by paying more attention on shared clauses.

References

1. Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 200–213, 2012.
2. Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette. An effective distributed d&c approach for the satisfiability problem. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 183–187, 2014.
3. Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezing and reactivating learnt clauses. In *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, pages 188–200, 2011.
4. Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pages 309–317, 2013.
5. Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Just-in-time compilation of knowledge bases. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.
6. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.
7. Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 197–205, 2014.
8. Tomas Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pages 156–172, 2015.
9. Anton Belov, Inês Lynce, and João Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012.
10. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
11. Armin Biere. *Bounded Model Checking*, chapter 14, pages 455–481. Volume 185 of Biere et al. [10], February 2009.
12. Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013*, page 51, 2013.
13. Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. In *proceedings of SAT competition.*, 2014.
14. Jingchao Chen. Minisat bcd and abcdsat: Solvers based on blocked clause decomposition. In *SAT RACE 2015 solvers description*, 2015.
15. Wahid Chrabakh and Richard Wolski. The gridsat portal: a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation: Practice and Experience*, 19(6):795–808, 2007.
16. Geoffrey Chu, Peter J. Stuckey, and AAron Harwood. Pminisat: a parallelization of minisat 2.0. Technical report, SAT Race, 2008.
17. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

18. Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2005.
19. Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.
20. Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 431–437, 1998.
21. Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 252–265, 2010.
22. Long Guo and Jean-Marie Lagniez. Dynamic polarity adjustment in a parallel SAT solver. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 67–73, 2011.
23. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 499–504, 2009.
24. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.
25. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 50–65, 2011.
26. Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, pages 372–386, 2010.
27. Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.
28. Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pages 276–292, 2013.
29. Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.
30. Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, 1996.
31. Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Improving search space splitting for parallel SAT solving. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 336–343, 2010.
32. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

33. Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 242–255, 2012.
34. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, pages 294–299, 2007.
35. Jussi Rintanen. *Planning and SAT*, chapter 15, pages 483–504. Volume 185 of Biere et al. [10], February 2009.
36. Olivier Roussel. ppfolio. http://www.cril.univ-artois.fr/ roussel/ppfolio.
37. SAT-race, 2015. http://baldur.iti.kit.edu/sat-race-2015/.
38. Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.
39. Alexander Semenov and Oleg Zaikin. Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 222–230, 2015.
40. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, pages 244–257, 2009.
41. Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, pages 475–476, 2012.
42. 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.