PaInleSS: a Framework for Parallel SAT Solving.

Ludovic Le Frioux[1,2]✉, Souheib Baarir[1,2,3], Julien Sopena[2], and Fabrice Kordon[2]

[1] LRDE, EPITA, Le Kremlin-Bicêtre, France
`firstname.name@lrde.epita.fr`
[2] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, Paris, France
CNRS, UMR 7606, LIP6, Paris, France
`firstname.name@lip6.fr`
[3] Université Paris Nanterre, France

Abstract. Over the last decade, parallel SAT solving has been widely studied from both theoretical and practical aspects. There are now numerous solvers that differ by parallelization strategies, programming languages, concurrent programming, involved libraries, etc.
Hence, comparing the efficiency of the theoretical approaches is a challenging task. Moreover, the introduction of a new approach needs either a deep understanding of the existing solvers, or to start from scratch the implementation of a new tool.
We present `PaInleSS`: a framework to build parallel SAT solvers for many-core environments. Thanks to its genericity and modularity, it provides the implementation of basics for parallel SAT solving like clause exchanges, Portfolio and Divide-and-Conquer strategies. It also enables users to easily create their own parallel solvers based on new strategies. Our experiments show that our framework compares well with some of the best state-of-the-art solvers.

Keywords: parallel, satisfiability, clause sharing, portfolio, cube and conquer

1 Introduction

Boolean satisfiability (SAT) has been used successfully in many contexts such as planning decision [19], hardware and software verification [6], cryptology [27] and computational biology [22], etc. This is due to the capability of modern SAT solvers to solve complex problems involving millions of variables and billions of clauses.

Most SAT solvers have long been sequential and based on the well-known DPLL algorithm [9, 8]. This initial algorithm has been dramatically enhanced by introducing sophisticated heuristics and optimizations: decision heuristics [29, 21], clauses learning [25, 33, 36], aggressive cleaning [2], lazy data structures [30], preprocessing [11, 24, 23], etc. The development of these enhancements has been greatly simplified by the introduction of `MiniSat` [10], an extensible SAT solver easing the integration of these heuristics in an efficient sequential solver.

The emergence of many-core machines opens new possibilities in this domain. Two classes of parallel techniques have been developed: competition-based (*a.k.a.,* Portfolio) and cooperation-based (*a.k.a.,* Divide-and-Conquer). In the Portfolio settings [14], many sequential SAT solvers compete for the solving of the whole problem. The first one to find a solution, or proving the problem to be unsatisfiable ends the computation. Divide-and-Conquer approaches use the guiding path method [35] to decompose, recursively and dynamically, the original problem in sub-problems that are solved separately by sequential solvers. In both approaches, sequential solvers can dynamically share learnt information. Many heuristics exist to improve this sharing by proposing trade-off between gains and overhead.

While the multiplication of strategies and heuristics provides perspectives for parallel SAT solving, it makes more complex development and evaluation of new proposals. Thus, any new contribution faces three main problems:

Problem 1: concurrent programming requires specific skills (synchronization, load balancing, data consistency, etc.). Hence, the theoretical efficiency of an heuristic may be annihilated by implementation choices.

Problem 2: most of the contributions mainly target a specific component in the solver while, to evaluate it, a complete one (either built from scratch or an enhancement of an existing one) must be available. This makes the implementation and evaluation of a contribution much harder.

Problem 3: an implementation, usually, only allows to test a single composition policy. Hence, it becomes hard to evaluate a new heuristic with different versions of the other mechanisms.

This paper presents PArallel INstantiabLE Sat Solver (`PaInleSS`)[4], a framework that simplifies the implementation and evaluation of new parallel SAT solvers for many-core environments. The components of `PaInleSS` can be instantiated independently to produce a new complete solver. The guiding principle is to separate the technical components dedicated to some specific aspect of concurrent programming, from the components implementing heuristics and optimizations embedded in a parallel SAT solver.

Our main contributions are the following:

- we propose a new modular and generic framework that can be used to implement new strategies with minimal effort and concurrent programming skills;
- we provide adaptors for some state-of-the-art sequential SAT solvers: `glucose` [2], `Lingeling` [5], `MiniSat` [10], and `MapleCOMSPS` [21].
- we show that it is easy to implemented strategies in `PaInleSS`, and provide some that are present in the classical solvers of the state-of-the-art: `glucose-syrup` [3], `Treengeling` [5], and `Hordesat` [4];
- we show the effectiveness of our modular design by instantiating, with a minimal effort, new original parallel SAT solver (by mixing strategies);

[4] painless.lrde.epita.fr

– we evaluate our approach on the benchmark of the parallel track of the SAT Race 2015. We compare the performance of solvers instantiated using the framework with the original solvers. The results show that the genericity provided by `PaInleSS` does not impact the performances of the generated instances.

The rest of the paper is organized as follows: Section 2 introduces useful background to deal with sequential SAT solving. Section 3 is dedicated to parallel SAT solving. Section 4 shows the architecture of `PaInleSS`. Section 5 presents different solvers implemented using `PaInleSS`. Section 6 analyzes the results of our experiments and Section 7 concludes and gives some perspectives. work.

2 About Sequential SAT Solving

In this section, after some preliminary definitions and notations, we introduce the most important features of modern sequential SAT solvers.

A *propositional variable* can have two possible values $\top$ (True) or $\bot$ (False). A *literal* l is a propositional variable (x) or its negation ($\neg x$). A *clause* ω is a finite disjunction of literals (noted $\omega = \bigvee_{i=1}^{k} \ell_i$). A clause with a single literal is called *unit clause*. A *conjunctive normal form (CNF) formula* φ is a finite conjunction of clauses (noted $\varphi = \bigwedge_{i=1}^{k} \omega_i$). For a given φ, the set of its variables is noted: V_φ. An assignment $\mathcal{A}$ of variables of φ, is a function $\mathcal{A} : V_\varphi \to \{\top, \bot\}$. $\mathcal{A}$ is total (complete) when all elements of V_φ have an image by $\mathcal{A}$, otherwise it is partial. For a given formula φ, and an assignment $\mathcal{A}$, a clause of φ is satisfied when it contains at least one literal evaluating to true, regarding $\mathcal{A}$. The formula φ is satisfied by $\mathcal{A}$ iff $\forall \omega \in \varphi, \omega$ is satisfied. φ is said to be SAT if there is at least one assignment that makes it satisfiable. It is defined as UNSAT otherwise.

Conflict Driven Clause Leaning. The majority of the complete state-of-the-art sequential SAT solvers are based on the Conflict Driven Clause Learning (CDCL) algorithm [25, 33, 36], that is an enhancement of the DPLL algorithm [9, 8]. The main components of a CDCL are depicted in Algorithm 1.

At each step of the main loop, `unitPropagation`[5] (line 4) is applied on the formula. In case of conflict (line 5), two situations can be observed: the conflict is detected at decision level 0 ($dl == 0$), thus the formula is declared UNSAT (lines 6-7); otherwise, a new asserting clause is derived by the conflict analysis and the algorithm backjumps to the assertion level [25] (lines 8-10). If there is no conflict (lines 11-13), a new decision literal is chosen (heuristically) and the algorithm continues its progression (adding a new decision level: $dl \leftarrow dl + 1$). When all variables are assigned (line 3), the formula is said to be SAT.

[5] The `unitPropagation` function implements the Boolean Constraint Propagation (BCP) procedure that forces (in cascade) the values of the variables in asserting clauses [8].

```
function CDCL()
    $dl \leftarrow 0$                                    // Current decision level
    while not all variables are assigned do
        $conflict \leftarrow$ unitPropagation()
        if $conflict$ then
            if $dl = 0$ then
                return $\perp$                              // $\varphi$ is UNSAT
            $\omega \leftarrow$ conflictAnalysis()
            addLearntClause($\omega$)
            $dl \leftarrow$ backjump($\omega$)
        else
            assignDecisionLiteral()
            $dl \leftarrow dl + 1$
    return $\top$                                     // $\varphi$ is SAT
```

Algorithm 1: CDCL algorithm.

The Learning Mechanism. The effectiveness of the CDCL lies in the *learning mechanism.* Each time a conflict is encountered, it is analyzed (`conflictAnalysis` function in Algorithm 1) in order to compute its reasons and to derive a *learnt clause.* While present in the system, this clause will avoid the same mistake to be made another time, and therefore allows faster deductions (conflicts/unit propagations).

Since the number of conflicts is very huge (in avg. 5000/s [2]), controlling the size of the database storing learnt clauses is a challenge. It can dramatically affect performance of the `unitPropagation` function. Many strategies and heuristics have been proposed to manage the cleaning of the stored clauses (*e.g.,* the Literal Block Distance (LBD) [2] measure).

With the two classical approaches used for parallel SAT solving: Portfolio and Divide-and-Conquer (see Section 3), multiple sequential solvers are used in parallel to solve the formula. With these paradigms sequential solvers can be seen as black boxes providing solving and clause sharing functionalities.

3 About Parallel SAT Solving

The arrival of many-core machines leads to new possibilities for SAT solving. Parallel SAT solving rely on two concepts: parallelization strategy and learnt clause exchanges. Two main parallelization methods have been developed: Portfolio and Divide-and-Conquer. We can also mention the hybrid approaches as alternatives, that are combinations of the first two techniques. With these parallelization strategies, it is possible to exchange learnt clauses, between the underling sequential solvers.

3.1 Parallelization Strategies

Portfolio. The Portfolio scheme has been introduced by [14], in `ManySat`. The main idea of this approach is to run sequential solvers working in parallel on the entire formula, in a competitive way. This strategy aims at increasing the probability of finding a solution using the *diversification* [12] (also known as swarming in others contexts) principle.

The diversification can only concern the used heuristics: several solvers (workers) with different heuristics are instantiated. They differ by their decision strategies, learning schemes, the used random seed, etc.

Another type of diversification, introduced in `HordeSat` [4], uses the *phase* of the variables: before starting the search each solver receives a special phase, acting as a soft division of the search space. Solvers are invited to visit a certain part of the search space but they can move out of this region during the search.

Another technique to ensure the diversification is the *block branching* [34]: each worker focuses on a particular subset (or block) of variables. Hence, the decision variables of a worker are chosen from the block it is in charge of.

Divide-and-Conquer. The Divide-and-Conquer approach is based on splitting the search space in disjoint parts. These parts are solved independently, in parallel, by different workers. As the parts are disjunct, if one of the partitions is proven to be SAT then the initial formula is SAT. The formula is UNSAT if all the partitions are UNSAT. The challenging points of the this method are: *dividing the search space* and *balancing jobs between workers*.

To divide the search space, the most used technique is based on the Shannon's decomposition, known as the *guiding path* [35]. The guiding path is a vector of literals (*a.k.a., cube*) that are assumed by the worker when solving the formula.

Choosing the best division variables is a hard problem requiring heuristics. If some parts are too easy this will lead to repeatedly divide the search space and ask for a new job (phenomenon known as *ping-pong effect*). As all the partitions do not require the same solving time, some workers may become idle and a mechanism for load balancing is needed. Each time a solver proves that its partition is UNSAT[6], it needs a new job. Another solver is chosen as target to divide its search space (*i.e.,* to extend its guiding path). The target will work on one of the new partition and the idle worker on the other one. This mechanism is often called *work stealing*.

Hybrid Approaches. As already presented, Portfolio and Divide-and-Conquer, are the two main explored approaches to parallelize SAT solving.

The Portfolio scheme is simple to implement, and uses the principle of diversification to increase the probability of solving the problem. However, since workers can overlap their search regions, the theoretical resulting speed-up is not as good as the one of the Divide-and-Conquer approach [17]. Surprisingly, while

[6] If the result is SAT the global solving ends.

giving a better theoretical speed-up, the Divide-and-Conquer approach suffers from the two challenging issues we mentioned: dividing the search space and balancing jobs between workers.

Emerging techniques, called *hybrid approaches*, propose to use simultaneously the two strategies, so that we benefit from the advantages of each, while trying to avoid their drawbacks.

A basic manner to mix the two approaches is to compose them. There are two possible strategies: Portfolio of Divide-and-Conquer (introduced by `c-sat` [31]), and Divide-and-Conquer of Portfolios (*e.g.,* `ampharos` [1] an adaptive Divide-and-Conquer that allows multiple workers on the same sub-part of the search space). Let us mention other more sophisticated ways to mix approaches like *scattering* [16, 18] or *transition heuristics* based strategies [7, 26, 1, 32].

3.2 Clauses Sharing

In all these parallelization paradigms, sharing information between workers is possible, the most important one being clauses learnt by each worker. Hence, the main questions are: *which clauses should be shared? And between which workers?* Indeed, sharing all clauses can have a bad impact on the overall behavior.

To answer the first question, many solvers rely on the standard measures, defined for sequential solvers (*i.e.,* activity, size, LBD): only clauses under a given threshold for these measures are shared. One simple way to get the threshold is to define it as constant it (*e.g.,* clauses up to size 8 are shared in `ManySat` [14]). More sophisticated approaches adapt thresholds dynamically in order to control the flow of shared clauses during the solving [13, 4].

A simple solution to the second question, adopted in almost all parallel SAT solvers, is to share clauses between all workers. However, a finer (but more complex) solution is to let each worker choose its emitters [20].

As a conclusion of this section, we can say that parallel SAT solving is based on two distinct concepts. First, there exist numerous strategies to parallelize SAT solving by organizing the workers search. Secondly, with all these strategies is it possible to share clauses between the workers. This two concepts have been our intuition sources for the design of the architecture of `PaInleSS`.

4 Architecture of the Framework

There exist numerous strategies to parallelize SAT solving, their performances heavily relying on their implementation. The most difficult issues deal with concurrent programming. Languages and libraries provide abstractions to deal with this difficulties, and according to these abstractions developers have more or less control on mechanisms such as memory or threads management (*e.g.,* `Java` vs `C++`). This will affect directly the performance of the produced solver.

Therefore, it is difficult to compare the strategies without introducing technological bias. Indeed, it is difficult to integrate new strategies on top of existing

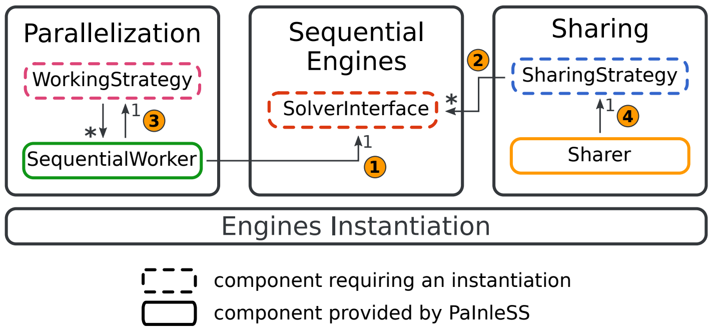


Fig. 1. Architecture of `PaInleSS`.

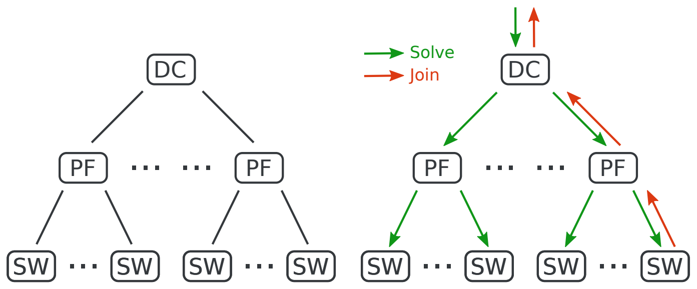
solvers, or to develop a new solver from scratch. Moreover, an implementation usually offers the possibility to modify a particular component, it is then difficult to test multiple combinations of components.

`PaInleSS` aims to be a solution to these problems. It is a generic, modular, and efficient framework, developed in `C++11`, allowing an easy implementation of parallel strategies. Taking black-boxed sequential solvers as input, it minimizes the effort to encode new parallelization and sharing strategies, thus enabling the implementation of complete SAT solvers at a reduced cost.

As mentioned earlier, a typical parallel SAT solver relies mainly on three core concepts: sequential engine(s), parallelization, and sharing. These last form the core of the `PaInleSS` architecture (see Fig. 1): the sequential engine is handled by the `SolverInterface` component. The parallelization is implemented by the `WorkingStrategy` and `SequentialWorker` components. Components `Sharing-Strategy` and `Sharer` are in charge of the sharing.

Sequential Engine. `SolverInterface` is an adapter for the basic functions expected from a sequential solver, it is divided in two subgroups: *solving* and *clauses export/import* (respectively represented by arrows `1` and `2` in Fig. 1). Subgroup `1` provides methods that interact with the solving process of the underling solver. The most important methods of this interface are:

- `SatResult solve(int[*] cube)`: tries to solve the formula, with the given cube (that can be empty in case of Portfolio). This method returns SAT, UNSAT, or UNKNOWN.
- `void setInterrupt()`: stops the current search initiated using the `solve` method.
- `void setPhase(int var, bool value)`: set the phase of variable `var` to `value`.
- `void bumpVariableActivity(int var, int factor)`: bumps `factor` times the activity of variable `var`.
- `void diversify()`: adjusts internal parameters of the solver, to diversify its behaviour.



(a) Tree structure (b) Workflow

Fig. 2. Example of a composed parallelization strategy.

Subgroup 2 provides methods to add/fetch learnt clauses to/from the solver:

- `void addClause(Clause cls)`: adds a permanent clause to the solver.
- `void addLearntClause(Clause cls)`: adds a learnt clause to the solver.
- `Clause getLearntClause()`: gets the oldest produced learnt clause from the solver.

The interface also provides methods to manipulate sets of clauses. The clauses produced or to be consumed by the solver, are stored in local *lockfree* queues (based on algorithm of [28]).

Technically, to integrate a new solver in `PaInleSS`, one needs to create a new class inheriting from `SolverInterface` and implement the required methods (*i.e.,* wrapping the methods of the API offered by the underlying solver). The framework currently provides some basic adaptors for `Lingeling` [5], `glucose` [2], `Minisat` [10], and `MapleCOMSPS` [21].

Parallelization. Basic parallelization strategies, such as those introduced in Section 3, must be implemented easily. We also aim at creating new strategies and mixing them.

A tree-structured (of arbitrary depth) composition mechanism enables the mix of strategies: internal nodes represent parallelization strategies, and leaves solvers. As an example (see Fig. 2(a)), a Divide-and-Conquer of Portfolios is represented by a tree of depth 3: the root corresponds to the Divide-and-Conquer having children representing the Portfolios acting on several solvers (the leaves of the tree).

`PaInleSS` implements nodes using the `WorkingStrategy` class, and leaves with the `SequentialWorker` class. This last is a subclass of `WorkingStrategy` that integrates an execution flow (a thread) operating the associated solver.

The overall solving workflow within this tree is user defined and guaranteed by the two main methods of the `WorkingStrategy` (arrows 3 in Fig. 1):

- `void solve(int[*] cube)`: according to the strategy implemented, this method manages the organization of the search by giving orders to the children strategies.

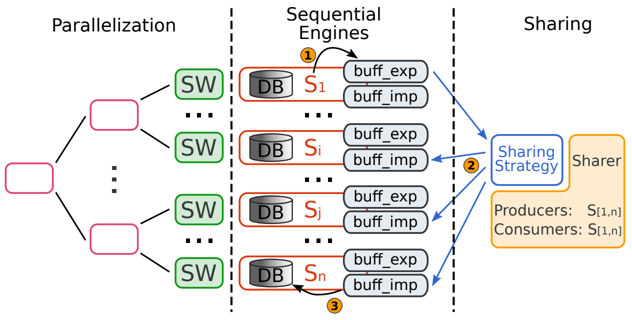


Fig. 3. Sharing mechanism implemented in `PaInleSS`.

- `void join(SatResult res, int[*] model)`: used to notify the parent strategy of the solving end. If the result is SAT, `model` will contain an assignment that satisfies the sub-formula treated by this node.

It is worth noting that the workflow must start by a call to the root's `solve` method and eventually ends by a call to the root's `join` method. The propagation of solving orders from a parent to one of its child nodes, is done by a call to the `solve` method of this last. The results are propagated back from a child to its parent by a call to the `join` method of this last. The solving can not be effective without a call to the leaves' `solve` methods.

Back to the example of Fig. 2(a). Consider the execution represented in Fig. 2(b). The solving order starts by a call to the root's (`DC` node) `solve` method. It is relayed trough the tree structure to the leaves (`SW` nodes). Here, once its problem is found SAT by one of the `SW`, it propagates back the result to its `PF` node parent via a call to the `join` method. According to the strategy of the `PF`, the `DC`'s `join` method is called and ends the global solving.

Hence, to develop its own parallelization strategy, the user should create one or more subclass of `WorkingStrategy` and to build the tree structure.

Sharing. In parallel SAT solving, we must pay a particular attention to the exchange of learnt clauses. Indeed, beside the theoretical aspects, a bad implementation of the sharing can dramatically impact the efficiency of the solver (*e.g.,* improper use of locks, synchronization problems). We now present how sharing is organized in `PaInleSS`.

When a solver learns a clause, it can share it according to a filtering policy such as the size or the LBD of the clause. To do so it puts the clause in a special buffer (`buff_exp` in Fig. 3). The sharing of the learnt clauses is realized by dedicated thread(s): `Sharer(s)`. Each one is in charge of a set of producers and consumers (these are references to `SolverInterface`). Its behaviour reduces to a loop of sleeping and exchange phases. This last is done by calling the interface of `SharingStrategy` class (arrow 4 in Fig. 1). The main method of this class is the following:

```
1  function main-PaInleSS (args: the program arguments)
2      solvers ← Create SolverInterface
3      root ← Create WorkingStrategy tree (solvers)
4      sharers ← Create SharingStrategy and Sharer (solvers)
5      root.solve()
6      while timeout or stop do
7          sleep(...)
8      print(root.getResult())
9      if root.getResult() == SAT then
10         print(root.getModel())
```

Algorithm 2: The main function of `PaInleSS`.

– `void doSharing(SolverInterface[*] producers, SolverInterface[*] consumers)`: according to the underlying strategy, this method gets clauses from the producers and add them to the consumers.

In the example of Fig. 3, the `Sharer` uses a given strategy, and all the solvers (S_i) are producers and consumers. The use of dedicated workflows (*i.e.,* threads) allows CPU greedy strategies to be run on a dedicated core, thus not interfering with the solving workers. Moreover, sharing phase can be done manipulating groups of clauses, allowing the use of more sophisticated heuristics. Finally, during its search a solver can get clauses from its import buffer (`buff_imp` in Fig. 3) to integrate them in its local database.

To define a particular sharing strategy the user only needs to provide a subclass of `SharingStrategy`. With our mechanism it is possible to have several groups of sharing each one manage by a `Sharer`. Moreover, solvers can be dynamically added/deleted from/to the producers and/or customers sets of a `Sharer`.

Engine Instantiation. To create a particular instance of `PaInleSS`, the user has to adapt the main function presented by Algorithm 2. The role of this function is to instantiate and bind all the components correctly. This actions are simplified by the use of parameters.

First, the concrete solver classes (inheriting from `SolverInterface`) are instantiated (line 2). Then the `WorkingStrategy` (including `SequentialWorker`) tree is implemented (line 3). This operation links `SequentialWorker` to their `SolverInterface`. Finally, the `Sharer`(s) and their concrete `SharingStrategy`(s) are created; the producers and consumers sets are initialized (line 4).

The solving starts by the call to the `solve` method of the root `WorkingStrategy` tree. The main thread will execute a loop, where it sleeps for an amount of time, and then checks if either the timeout has been reached or the solving ended (lines 6-7). It prints the final result (line 8), plus the model in case of a SAT instance (lines 9-10).

5 Implementing and Combining Existing Strategies

To validate the generic aspect of our approach, we selected three efficient state-of-the-art parallel SAT solvers: `glucose-syrup` [3], `Treengeling` [5], and `Hordesat` [4]. For each selected solver, we implemented a solver that mimics the original one using `PaInleSS`. To show the modularity of `PaInleSS`, we used the already developed components to instantiate two new original solvers that combine existing strategies.

Solver "à la glucose-syrup". The `glucose-syrup`[7] solver is the winner of the parallel track of the SAT Race 2015. It is a Portfolio based on the sequential solver `glucose` [2]. The sharing strategy exchanges all the exported clauses between all the workers. Beside, the workers have customized settings in order to diversify their search.

Hence, implementing a solver "à la `glucose-syrup`", namely `PaInleSS-glucose-syrup`, required the following components: `Glucose` an adaptor to use the `glucose` solver; `Portfolio` a simple `WorkingStrategy` that implements a Portfolio strategy; `SimpleSharing` a `SharingStrategy` that exchanges all the exported clauses from the producers to all the consumers with no filtering.

The implementation of `PaInleSS-glucose-syrup` required 355 lines of code (LoC) for the adaptor, 95 LoC for the Portfolio, and 44 LoC for the sharing strategy.

Solver "à la Treengeling". The `Treengeling`[8] solver is the winner of the parallel track of the SAT Competition 2016. It is based on the sequential engine `Lingeling` [5]. Its parallelization strategy is a variant of Divide-and-Conquer called Cube-and-Conquer [15]. The solving is organized in rounds. Some workers search for a given number of conflicts. When the limit is reached, some are selected to split their sub-spaces using a lookahead heuristic. The sharing is restricted to the exchange of unit clauses from a special worker. This last is also in charge of the solving of the whole formula during all the execution.

To implement a solver "à la `Treengeling`", namely `PaInleSS-treengeling`, we needed the following components: `Lingeling`, an adaptor of the sequential solver `Lingeling`; `CubeAndConquer` a `WorkingStrategy`, that implements a Cube-and-Conquer [15]; `SimpleSharing` already used to define for the `glucose-syrup` like solver. In this case, the underlying sequential solvers are parametrized to export only unit clauses, and only the special worker is a producer.

For the `CubeAndConquer` we choose time to manage rounds because it allows, once one worker has encountered an UNSAT situation, to restart the worker with another guiding-path. In the original implementation, rounds are managed using numbers of conflicts, this makes the reuse of idle CPU much harder.

The implementation of `PaInleSS-treengeling` needed 377 LoC for the adaptor and 249 LoC for `CubeAndConquer`.

[7] www.labri.fr/perso/lsimon/downloads/softwares/glucose-syrup.tgz
[8] www.fmv.jku.at/lingeling/lingeling-bbc-9230380-160707.tar.gz

Solver “à la Hordesat”. `Hordesat`[9] is a Portfolio-based solver with a modular design. `Hordesat` uses as sequential engine either `Minisat` [10] or `Lingeling`. It is a Portfolio where the sharing is realized by controlling the flow of exported clauses. Every second, 1500 literals (*i.e.,* sum of the size of the clauses) are exported from each sequential engine. Moreover, we used the `Lingeling` solver and the native diversification of `Plingeling` [5] (a Portfolio solver of `Lingeling`) combined to the random sparse diversification (presented as the best combination by [4]).

The solver “à la `Hordesat`”, namely `PaInleSS-hordesat`, required the following components: `Lingeling` and `Portfolio` that have been implemented earlier; `HordesatSharing` a `SharingStrategy` that implements the `Hordesat` sharing strategy. This last required only 148 LoC.

Combining Existing Strategies. Based on the implemented solvers, we reused the obtained components to quickly build two new original solvers.

`PaInleSS-treengeling-hordesat`: it is a `PaInleSS-treengeling`-based solver that shares clauses using the strategy of `Hordesat`. The implementation of this solver reuses the `Lingeling`, `CubeAndConquer`, and `HordesatSharing` classes. To instantiate this solver we only needed a special parametrization. Beside, the modularity aspects, by this instantiation, we aimed to investigate the impact of a different sharing strategy on the overall performances of `PaInleSS-treengeling`.

`PaInleSS-treengeling-glucose`: it is a Portfolio solver that mixes Cube-and-Conquer of `Lingeling`, and a Portfolio of `Glucose` solvers. Here, `Glucose` workers export unit and *glue* clauses [2] (*i.e.,* clauses with LBD equals to 2) to the other solvers. This last solver reuses the following components: `Lingeling`, `Glucose`, `Portfolio`, `CubeAndConquer`, `SimpleSharing`. Only 15 LoC are required to build the parallelization strategy tree. By the instantiation of this solver, we aimed to study the effect of mixing some parallelisation strategies.

6 Numerical Results

This section presents the results of experiments we realized using the solvers described in Section 5: `PaInleSS-glucose-syrup`, `PaInleSS-treengeling`, `PaInleSS-hordesat`, `PaInleSS-treengeling-hordesat`, and `PaInleSS-treengeling-glucose`. The goal here is to show that the introduction of genericity does not add an overhead *w.r.t.* the original solvers.

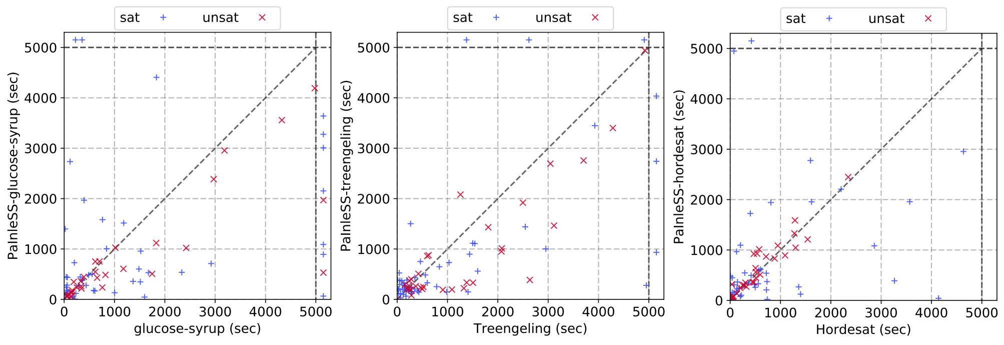
All the experiments have been executed on a parallel machine with 40 processors Intel Xeon CPU E7- 2860 @ 2.27GHz, and 500 Go of memory. We used the 100 instances of the parallel track of the SAT Race 2015[10]. All experiments have been conducted using the following parametrisations: each solver has been run once on each instance, with a time-out of 5000 seconds (as in the SAT Race). We limited the number of involved CPUs to 36.

[9] baldur.iti.kit.edu/hordesat/files/hordesat.zip

[10] baldur.iti.kit.edu/sat-race-2015/downloads/sr15bench-hard.zip

Table 1. Results of the different solvers. The different columns represent: the number of UNSAT solved instances, SAT solved instances, total solved instances, and the cumulative time spent solving the instances solved by the two solvers.

Solver	UNSAT	SAT	Total	Cum. Time Inter.
`glucose-syrup`	30	41	71	15h37
`PaInleSS-glucose`	**32**	**46**	**78**	**13h18**
`Treengeling`	32	50	82	20h55
`PaInleSS-treengeling`	32	50	82	**14h12**
`Hordesat`	31	**44**	**75**	15h05
`PaInleSS-hordesat`	31	43	74	**14h19**

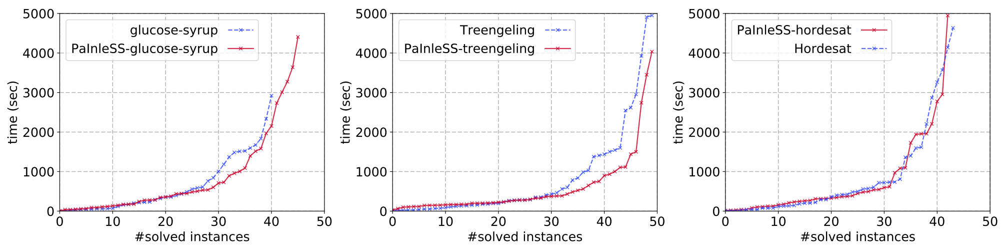


(a) `glucose-syrup` (b) `Treengeling` (c) `Hordesat`

Fig. 4. Scatter plots of `PaInleSS`'s solvers against state-of-the-art ones.

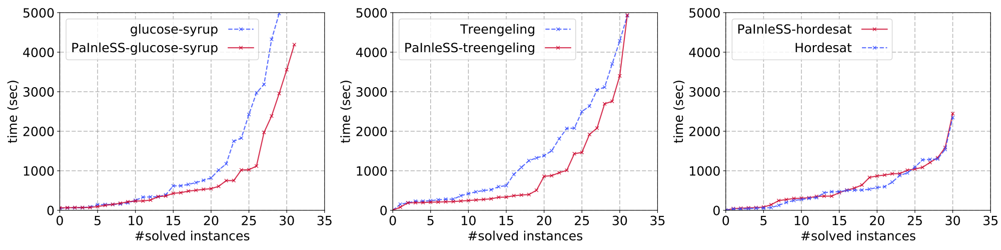
The number of solved instances per solver are reported in Table 1. Globally, these primary results show that our solvers compare well to the studied state-of-the-art solvers. We can deduce that the genericity offered by `PaInleSS` does not impact the global performances. Moreover, on instances solved by both, the original solver and our implementation, the cumulative solving time is in our favor (see column Cum. Time. Inter. in Table 1). A more detailed analysis is given for each solver in the rest of the section.

`PaInleSS-glucose-syrup` *vs.* `glucose-syrup`. Our implementation of the `glucose-syrup` parallelization strategy was able to solve 7 more instances compared to `glucose-syrup`. This concerns both SAT and UNSAT instances as shown in the scatter plot of Fig. 4(a)) and, in the cactus plots of Fig. 5(a) and 6(a). This gain is due to our careful design of the sharing mechanism that is decentralized and uses lock-free buffers. Indeed in `glucose-syrup` a global buffer is used to exchange clauses, which requires import/export to use a unique lock, thus introducing a bottleneck. The absence of bottleneck in our implementation increases the parallel all over the execution, explaining our better performances.



(a) `glucose-syrup` (b) `Treengeling` (c) `Hordesat`

Fig. 5. Cactus plots of SAT instances of `PaInleSS`'s solvers against state-of-the-art ones.



(a) `glucose-syrup` (b) `Treengeling` (c) `Hordesat`

Fig. 6. Cactus plots of UNSAT instances of `PaInleSS`'s solvers against state-of-the-art ones.

`PaInleSS-treengeling` *vs.* `Treengeling`. Concerning `Treengeling`, our implementation has comparable results. Fig. 4(b) shows that the average solving time of SAT instances is quite similar, while for the UNSAT instances, our implementation is in average faster. This is corroborated by the cactus plot depicted in Fig. 6(b). This speed up is due to our fine implementation of the Cube-and-Conquer strategy, thus increasing the real parallelism all over the execution and explaining our better performances on UNSAT instances.

`PaInleSS-hordesat` *vs.* `Hordesat`. Although `Hordesat` was able to solve 1 more instance than our tool, results are comparable. Moreover scatter plot of Fig. 4(c), and cactus plots of Fig. 6(c) and Fig. 5(c) exhibit quit similar results for the two tools. For instances solved by both tools, our tool was a beat faster and used almost 3000 seconds less as pointed out in Table 1. As the sharing strategy of `Hordesat` is mainly based on two parameters, namely the number of exchanged literals per round, and the sleeping time of sharer by round, we think that a finer tuning of this couple of parameters for our implementation could improve the performances of our tool.

Results of the Composed Solvers. `PaInleSS-treengeling-hordesat` solved 81 instances (49 SAT and 32 UNSAT), and `PaInleSS-treengeling-glucose` solved

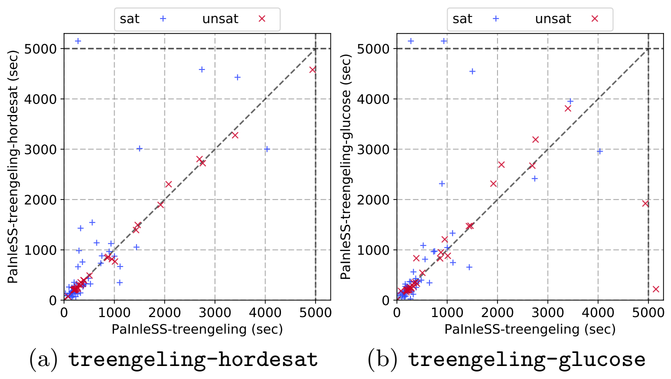


Fig. 7. Scatter plots of the composed solvers against `PaInleSS-treengeling`.

81 instances (48 SAT and 33 UNSAT). The scatter plot of the two strategies (Fig. 7), show that these strategies are almost equivalent w.r.t. the original ones. These results allow us to conclude that the introduced strategies do not add any value to the original one.

7 Conclusion

Testing and implementing new strategies for parallel SAT solving has become a challenging issue. Any new contribution in the domain faces the following problems: concurrent programming requires specific skills, testing new strategies required a prohibitive development of a complete solver (either built from scratch or an enhancement of an existing one), an implementation often allows to test only a single composition policy and avoids the evaluation of a new heuristic with different versions of the other mechanisms.

To tackle these problems we proposed `PaInleSS`, a modular, generic and efficient framework for parallel SAT solving. We claimed that its modularity and genericity allow the implementation of basic strategies, as well as new onces and their combination with a minimal effort and concurrent programming skills.

We have proven our claims, first, by the implementation of strategies present in some state-of-the-art solvers: `glucose-syrup`, `Treengeling`, and `Hordesat`. Second, we reused the developed complements to derive, easily, new solvers that mix strategies. We also show that the instantiated solvers are as efficient as the original one (and even better), by conducting a set experiments using benchmarks of the SAT Race 2015.

As perspectives, we plan to adapt our framework for mutli-machine environments. We also would like to enhance `PaInleSS` with helpful tools to monitor algorithm metrics (*e.g.,* number of shared clauses), system metrics (*e.g.,* synchronization time, load balancing), and to facilitate the debugging work. Another interesting point is the simplification of the instantiation mechanism by providing a domain specific language (DSL).

References

1. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel sat solver. In: int. conf. on Principles and Practice of Constraint Programming. pp. 30–48. Springer (2016)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: IJCAI. vol. 9, pp. 399–404 (2009)
3. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel sat solvers. In: int. conf. on Theory and Applications of Satisfiability Testing. pp. 197–205. Springer (2014)
4. Balyo, T., Sanders, P., Sinz, C.: Hordesat: A massively parallel portfolio sat solver. In: int. conf. on Theory and Applications of Satisfiability Testing. pp. 156–172. Springer (2015)
5. Biere, A.: Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. SAT COMPETITION 2016 p. 44 (2016)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. Tools and Algorithms for the Construction and Analysis of Systems pp. 193–207 (1999)
7. Blochinger, W.: Towards robustness in parallel sat solving. In: PARCO. pp. 301–308 (2005)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (Jul 1962)
9. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (Jul 1960)
10. Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
11. En, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: Theory and Applications of Satisfiability Testing. pp. 61–75. Springer (2005)
12. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel sat solving. In: int. conf. on principles and practice of constraint programming. pp. 252–265. Springer (2010)
13. Hamadi, Y., Jabbour, S., Sais, J.: Control-based clause sharing in parallel sat solving. In: Autonomous Search, pp. 245–267. Springer (2011)
14. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262 (2009)
15. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl sat solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
16. Hyvärinen, A.E., Junttila, T., Niemelä, I.: A distribution method for solving sat in grids. In: on theory and applications of satisfiability testing. pp. 430–435. Springer (2006)
17. Hyvärinen, A.E., Junttila, T., Niemelä, I.: Partitioning search spaces of a randomized search. In: Congress of the Italian Association for Artificial Intelligence. pp. 243–252. Springer (2009)
18. Hyvärinen, A.E., Manthey, N.: Designing scalable parallel sat solvers. In: int. conf. on Theory and Applications of Satisfiability Testing. pp. 214–227. Springer (2012)
19. Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: ECAI. vol. 92, pp. 359–363 (1992)

20. Lazaar, N., Hamadi, Y., Jabbour, S., Sebag, M.: Cooperation control in Parallel SAT Solving: a Multi-armed Bandit Approach. Research Report RR-8070, INRIA (Sep 2012)
21. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for sat solvers. In: Theory and Applications of Satisfiability Testing. pp. 123–140. Springer (2016)
22. Lynce, I., Marques-Silva, J.: Sat in bioinformatics: Making the case with haplotype inference. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 136–141. Springer (2006)
23. Manthey, N.: Coprocessor 2.0–a flexible cnf simplifier. In: int. conf. on Theory and Applications of Satisfiability Testing. pp. 436–441. Springer (2012)
24. Manthey, N.: Coprocessor–a standalone sat preprocessor. In: Applications of Declarative Programming and Knowledge Management, pp. 297–304. Springer (2013)
25. Marques-Silva, J.P., Sakallah, K., et al.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. on Computers 48(5), 506–521 (1999)
26. Martins, R., Manquinho, V., Lynce, I.: Improving search space splitting for parallel sat solving. In: IEEE int. conf. on Tools with Artificial Intelligence. vol. 1, pp. 336–343. IEEE (2010)
27. Massacci, F., Marraro, L.: Logical cryptanalysis as a sat problem. Journal of Automated Reasoning 24(1), 165–203 (2000)
28. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 267–275. ACM (1996)
29. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
30. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
31. Ohmura, K., Ueda, K.: c-sat: A parallel sat solver for clusters. In: Theory and Applications of Satisfiability Testing, pp. 524–537. Springer (2009)
32. Schulz, S., Blochinger, W.: Cooperate and compete! a hybrid solving strategy for task-parallel sat solving on peer-to-peer desktop grids. In: High Performance Computing and Simulation. pp. 314–323. IEEE (2010)
33. Silva, J.P.M., Sakallah, K.A.: Graspa new search algorithm for satisfiability. In: IEEE/ACM int. conf. on Computer-aided design. pp. 220–227. IEEE (1997)
34. Sonobe, T., Inaba, M.: Portfolio with block branching for parallel sat solvers. In: int. conf. on Learning and Intelligent Optimization. pp. 247–252. Springer (2013)
35. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21(4), 543–560 (1996)
36. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: IEEE/ACM int. conf. on Computer-aided design. pp. 279–285. IEEE Press (2001)