Modular and Efficient Divide-and-Conquer SAT Solver on Top of the Painless Framework

Ludovic Le Frioux[1,2]✉, Souheib Baarir[2,3], Julien Sopena[2,4], and Fabrice Kordon[2]

[1] LRDE, EPITA, F-94043, Le Kremlin-Bicêtre
`ludovic@lrde.epita.fr`
[2] Sorbonne Université, CNRS, LIP6, UMR 7606, F-75005, Paris
`firstname.name@lip6.fr`
[3] Université Paris Lumières, Université Paris Nanterre, F-92000, Nanterre
[4] Inria, DELYS Team, F-75005, Paris

Abstract. Over the last decade, parallel SATisfiability solving has been widely studied from both theoretical and practical aspects. There are two main approaches. First, divide-and-conquer (`D&C`) splits the search space, each solver being in charge of a particular subspace. The second one, portfolio launches multiple solvers in parallel, and the first to find a solution ends the computation. However although `D&C` based approaches seem to be the natural way to work in parallel, portfolio ones experimentally provide better performances.
An explanation resides on the difficulties to use the native formulation of the SAT problem (*i.e.,* the CNF form) to compute an *a priori* good search space partitioning (*i.e.,* all parallel solvers process their subspaces in comparable computational time). To avoid this, dynamic load balancing of the search subspaces is implemented. Unfortunately, this is difficult to compare load balancing strategies since state-of-the-art SAT solvers appropriately dealing with these aspects are hardly adaptable to various strategies than the ones they have been designed for.
This paper aims at providing a way to overcome this problem by proposing an implementation and evaluation of different types of divide-and-conquer inspired from the literature. These are relying on the `Painless` framework, which provides concurrent facilities to elaborate such parallel SAT solvers. Comparison of the various strategies are then discussed.

Keyword. Divide-and-conquer, parallel satisfiability, tool

1 Introduction

Modern SAT solvers are now able to handle complex problems involving millions of variables and billions of clauses. These tools have been used successfully to solve constraints' systems issued from many contexts, such as planning decision [16], hardware and software verification [7], cryptology [23], and computational biology [20], etc.

State-of-the-art complete SAT solvers are based on the well-known Conflict-Driven Clause Learning (CDCL) algorithm [21,28,30]. With the emergence of many-core machines, multiple parallelisation strategies have been conducted on these solvers. Mainly, two classes of parallelisation techniques have been studied: divide-and-conquer (`D&C`) and portfolio. Divide-and-conquer approaches, often based on the guiding path method, decompose recursively and dynamically, the original search space in subspaces that are solved separately by sequential solvers [29,12,14,1,2,26]. In the portfolio setting, many sequential SAT solvers compete for the solving of the whole problem [11,4,5]. The first to find a solution, or proving the problem to be unsatisfiable ends the computation. Although divide-and-conquer approaches seem to be the natural way to parallelise SAT solving, the outcomes of the parallel track in the annual SAT Competition show that the best state-of-the-art parallel SAT solvers are portfolio ones.

The main problem of divide-and-conquer based approaches is the search space division so that load is balanced over solvers, which is a theoretical hard problem. Since no optimal heuristics has been found, solvers compensate non optimal space division by enabling dynamic load balancing. However, state-of-the-art SAT solvers appropriately dealing with these aspects are hardly adaptable to various strategies than the ones they have been designed for [1,6,2]. Hence, it turns out to be very difficult to make fair comparisons between techniques (*i.e.,* using the same basic implementation). Thus, we believe it is difficult to conclude on the (non-) effectiveness of a technique with respect to another one and this may lead to premature abortion of potential good ideas.

This paper tries to solve these problems by proposing a simple, generic, and efficient divide-and-conquer component on top of the `Painless` [18] framework. This component eases the implementation and evaluation of various strategies, without any compromise on efficiency. Main contributions of this paper are the followings:

- an overview of state-of-the-art divide-and-conquer methods;
- a complete divide-and-conquer component that has been integrated to the `Painless` framework;
- a fair experimental evaluation of different types of divide-and-conquer inspired from the literature, and implemented using this component.

These implementations have often similar and sometimes better performances compared with state-of-the-art divide-and-conquer SAT solvers.

Let us outline several results of this work. First, our `Painless` framework is able to support implementation of multiple `D&C` strategies in parallel solvers. Moreover, we have identified "axes" for customization and adaptation of heuristics. Thus, we foresee it will be much easier to explore next `D&C` strategies. Second, our best implementation at this stage is comparable in terms of performance, with the best state-of-the-art `D&C` solvers, which shows our framework's efficiency.

This paper is organized as follows: Section 2 introduces useful background to deal with the SAT problem. Section 3 is dedicated to divide-and-conquer based parallel SAT solving. Section 4 explains the mechanism of divide-and-conquer we have implemented in `Painless`. Section 5 analyses the results of our experiments, and Section 6 concludes and gives some perspectives.

2 Background

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 formula 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.

Algorithm 1: CDCL algorithm

```
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 $\bot$                               // $\varphi$ is UNSAT
            end
            $\omega \leftarrow$ conflictAnalysis()
            addLearntClause($\omega$)
            $dl \leftarrow$ backjump($\omega$)
        end
        else
            assignDecisionLiteral()
            $dl \leftarrow dl + 1$
        end
    end
    return $\top$                                           // $\varphi$ is SAT
```

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 [21,28,30], that is an enhancement of the DPLL algorithm [10,9]. The main components of a CDCL are presented 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

[5] The `unitPropagation` function implements the Boolean Constraint Propagation (BCP) procedure that forces (in cascade) the values of the variables in unit clauses [9].

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 [21] (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.

The Learning Mechanism. The effectiveness of the CDCL lies in the *learning mechanism* (line 10). Each time a conflict is encountered, it is analyzed (`conflictAnalysis` function in Algorithm 1) in order to compute its reasons and 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 [3]), 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) [3] measure).

3 Divide-and-Conquer based Parallel SAT Solvers

The divide-and-conquer strategy in parallel SAT solving is based on splitting the search space into subspaces that are submitted to different workers. If a subspace is proven SAT then the initial formula is SAT. The formula is UNSAT if all the subspaces are UNSAT. The challenging points of the divide-and-conquer mechanism are: *dividing the search space*, *balancing jobs between workers*, and *exchanging learnt clauses*.

3.1 Dividing the Search Space

This section describes how to create multiple search subspaces for the studied problem, and the heuristics to balance their estimated computational costs.

Techniques to Divide the Search Space. To divide the search space, the most often used technique is the *guiding path* [29]. It is a conjunction of literals (called cube) that are assumed by the invoked solver (worker). Let φ be a formula, and $x \in \mathcal{V}_\varphi$ a variable. Thanks to Shannon decomposition, we can rewrite φ as $\varphi = (\varphi \wedge x) \vee (\varphi \wedge \neg x)$. The two guiding paths here are reduced to a single literal: (x) and $(\neg x)$. This principle can be applied recursively on each subspaces to create multiple guiding paths.

Figure 1 illustrates such an approach where six subspaces have been created from the original formula. They are issued from the following guiding paths: $(d \wedge b)$, $(d \wedge \neg b)$, $(\neg d \wedge a \wedge b)$, $(\neg d \wedge a \wedge \neg b)$, $(\neg d \wedge \neg a \wedge x)$, $(\neg d \wedge \neg a \wedge \neg x)$. The

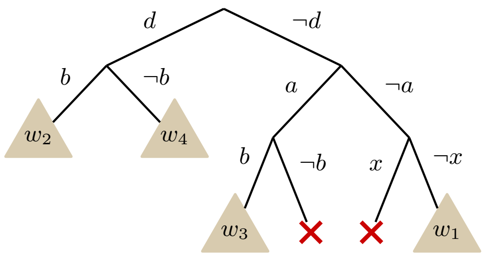


Fig. 1: Using guiding path to divide the search space

subspaces that have been proven UNSAT, are highlighted with red crosses. The rest of the subspaces are submitted to workers (noted w_i).

It is worth noting that other partitioning techniques exist that were initially developed for distributed systems rather than many-cores machines. We can cite the *scattering* [13], and the *xor partitioning* [27] approaches.

Choosing Division Variables. Choosing the best *division variable* is a hard problem, requiring the use of heuristics. A good division heuristic should decrease the overall total solving time[6]. Besides, it should create balanced subspaces w.r.t. their solving time: if some subspaces are too easy to solve this will lead to repeatedly asking for new jobs and redividing the search space (phenomenon known as *ping pong effect* [15]).

Division heuristics can be classified in two categories: *look ahead* and *look back*. Look ahead heuristics rely on the possible future behaviour of the solver. Contrariwise, look back heuristics rely on statistics gathered during the past behaviour of the solver. Let us present the most important ones.

Look ahead. In stochastic SAT solving (chapters 5 and 6 in [8]), look ahead heuristics are used to choose the variable implying the biggest number of unit propagations as a decision variable. When using this heuristic for the division, one tries to create the smallest possible subspaces (*i.e.,* with the least unassigned variables). The main difficulty of this technique is the generated cost of applying the unit propagation for the different variables. The so-called "cube-and-conquer" solver presented in [12] relies on such an heuristic.

Look back. Since sequential solvers are based on heuristics to select their decision variables, these can naturally be used to operate the search space division. The idea is to use the variables' VSIDS-based [25] order[7] to decompose the search in subspaces. Actually, when a variable is highly ranked w.r.t. to this order, then it is commonly admitted that it is a good starting point for a separate exploration [13,22,2].

[6] Compared to the solving time using a sequential solver
[7] The number of their implications in propagation conflicts.

Another explored track is the number of *flips* of the variables [1]. A flip is when a variable is propagated to the reverse of its last propagated value. Hence, ranking the variables according to the number of their flips, and choosing the highest one as a division point helps to generate search subspaces with comparable computational time. This can be used to limit the number of variables on which the look ahead propagation is applied by preselecting a predefined percentage of variables with the highest number of flips.

Another look back approach, called propagation rate (PR), tends to produce the same effect as the look ahead heuristics [26]. The PR of a variable v is the ratio between the numbers of propagations due to the branching of v divided by the number of time v has been chosen as a decision. The variable with the highest PR is chosen as division point.

3.2 Load Balancing

Despite all the effort to produce balanced subspaces, it is practically impossible to ensure the same difficulty for each of them. Hence, some workers often become quickly idle, thus requiring a dynamic load balancing mechanism.

A first solution to achieve dynamic load balancing is to rely on *work stealing*: each time a solver proves its subspace to be UNSAT[8], it asks for a new job. A target worker is chosen to divide its search space (*e.g.,* extends its guiding path). Hence, the target is assigned to one of the new generated subspaces, while the idle solver works on the other. The most common architecture to implement this strategy is based on a master/slave organization, where slaves are solvers.

When a new division is needed, choosing the best target is a challenging problem. For example, the `Dolius` solver [1] uses a FIFO order to select targets: the next one is the worker that is working for the longest time on its search space. This strategy guarantees fairness between workers. Moreover the target has a better knowledge of its search space, resulting in a better division when using a look back heuristic.

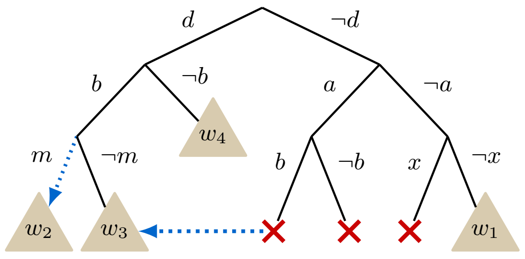


Fig. 2: Load balancing through work stealing

[8] If the result is SAT the global resolution ends.

Let us suppose in the example of Figure 1 that worker w_3 proves its subspace to be UNSAT, and asks for a new one. Worker w_2 is chosen to divide and share its subspace. In Figure 2, m is chosen as division variable and two new guiding paths are created, one for w_2 and one for w_3. Worker w_3 now works on a new subspace and its new guiding path is $(d \wedge b \wedge \neg m)$, while the guiding path of w_2 is $(d \wedge b \wedge m)$.

Another solution to perform dynamic load balancing is to create more search subspaces (jobs) than available parallel workers (cube-and-conquer [12]). These jobs are then managed via a work queue where workers pick new jobs. To increase the number of available jobs at runtime, a target job is selected to be divided. The strategy implemented in `Treengeling` [6] is to choose the job with the smallest number of variables; this favours SAT instances.

3.3 Exchanging Learnt Clauses

Dividing the search space can be subsumed to the definition of constraints on the values of some variables. Technically, there exist two manners to implement such constraints: (i) constrain the original formula; (ii) constrain the decision process initialisation of the used solver.

When the search space division is performed using (i), some learnt clauses cannot be shared between workers. This is typically the case of learnt clauses deduced from at least one clause added for space division, otherwise, correctness is not preserved. The simplest solution to preserve correctness is then to disable clause sharing [6]. Another (more complex) approach is to mark the clauses that must not be shared [17]. Clauses added for the division are initially marked. Then, the tag is propagated to each learnt clause that is deduced from at least one already marked clause.

When the search space division is performed using (ii), some decisions are forced. With this technique there is no sharing restrictions for any learnt clauses. This solution is often implemented using the assumption mechanisms [1,2].

4 Implementation of a Divide-and-Conquer

This section presents the divide-and-conquer component we have built on top of the `Painless` framework. First, we recall the general architecture and operations of `Painless`. Then we describe the generic divide-and-conquer component's mechanisms. Finally we detail the different heuristics we have instantiated using this component.

4.1 About the Painless Framework

`Painless` [18] is a framework that aims at simplifying the implementation and evaluation of parallel SAT solvers for many-core environments. Thanks to its genericity and modularity, the components of `Painless` can be instantiated independently to produce new complete solvers.

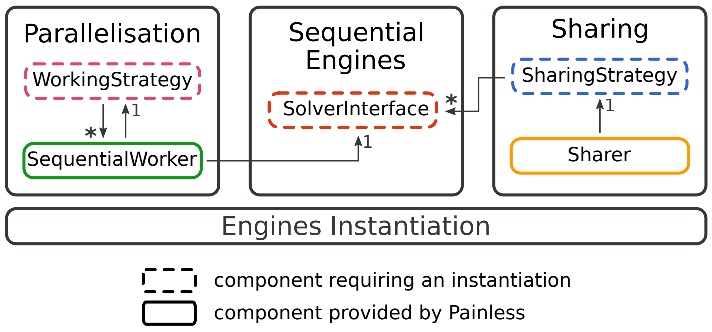


Fig. 3: Architecture of `Painless`

The main idea of the framework is to separate the technical components (*e.g.,* those dedicated to the management of concurrent programming aspects) from those implementing heuristics and optimizations embedded in a parallel SAT solver. Hence, the developer of a (new) parallel solver concentrates his efforts on the functional aspects, namely parallelisation and sharing strategies, thus delegating implementation issues (*e.g.,* data concurrent access protection mechanisms) to the framework.

Three main components arise when treating parallel SAT solvers: *sequential engines*, *parallelisation*, and *sharing*. These form the global architecture of `Painless` depicted in Figure 3.

Sequential Engines. The core element considered in the framework is a sequential SAT solver. This can be any CDCL state-of-the art solver. Technically, these engines are operated through a generic interface providing basics of sequential solvers: *solve, interrupt, add clauses*, etc.

Thus, to instantiate `Painless` with a particular solver, one needs to implement the interface according to this engine.

Parallelisation. To build a parallel solver using the aforementioned engines, one needs to define and implement a parallelisation strategy. Portfolio and divide-and-conquer are the basic known ones. Also, they can be arbitrarily composed to form new strategies.

In `Painless`, a strategy is represented by a tree-structure of arbitrarily depth. The internal nodes of the tree represent parallelisation strategies, and leaves are core engines. Technically, the internal nodes are implemented using `WorkingStrategy` component and the leaves are instances of `SequentialWorker` component.

Hence, to develop its own parallelisation strategy, the user should create one or more strategies, and build the required tree-structure.

Sharing. In parallel SAT solving, the exchange of learnt clauses warrants a particular focus. Indeed, besides the theoretical aspects, a bad implementation of a good sharing strategy may dramatically impact the solver's efficiency.

In `Painless`, solvers can export (import) clauses to (from) the others during the resolution process. Technically, this is done by using lockfree queues [24]. The sharing of these learnt clauses is dedicated to particular components called `Sharers`. Each `Sharer` in charge of sets of producers and consumers and its behaviour reduces to a loop of sleeping and exchange phases.

Hence, the only part requiring a particular implementation is the exchange phase, that is user defined.

4.2 The Divide-and-Conquer Component in Painless

To implement divide-and-conquer solvers with `Painless`, we define a new component. It is based on a master/slaves architecture.

Figure 4 shows the architecture of our tool. It contains several entities. The *master* is a thread executing the only `D&C` instance of the `WorkingStrategy` class. The *workers* are slave threads executing instances of the `SequentialWorker` class. An instance of the `Sharing` class allows workers to share clauses.

The master and the workers interact asynchronously by means of events. In the initialisation phase, the master may send asynchronous events to himself too.

Master. The master (1) initialises the `D&C` component; (2) selects targets to divide their search spaces; (3) and operates the division along with the relaunch of the associated solvers. These actions are triggered by the events `INIT`, `NEED_JOB`, and `READY_TO_DIV`, respectively. In the remainder of this section we consider a configuration with N workers.

The master can be in two states: either it is sleeping, or it is currently processing an incoming event. Initially the master starts a first solver on the whole formula by sending it the `SOLVE` event. It then generates $N-1$ `NEED_JOB` events to himself. This will provoke the division of the search space in N subspaces according the to implemented policy. At the end of this initialisation phase, it returns to its sleeping state. At this point, all workers are processing their subspaces.

Each time a worker needs a job, it notifies the master with a `NEED_JOB` event.

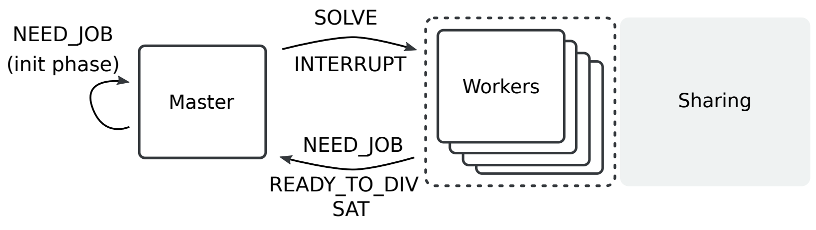


Fig. 4: Architecture of the divide-and-conquer SAT solver

All over its execution, the master reacts to the `NEED_JOB` event as follows:

1. it selects a target using the current policy[9], and requests this target to interrupt by sending an `INTERRUPT` event. Since this is an asynchronous communication, the master may process other events until it receives a `READY_TO_DIV` event;
2. once it receives a `READY_TO_DIV` event, the master proceeds to the effective division of the subspace of the worker which emitted the event. Both the worker which emitted the event and the one which requested a job are then invited to solve their new subspaces through the send of a `SOLVE` event.

The master may receive a `SAT` event from its workers. It means a solution has been computed and the whole execution must end. When a worker ends in an UNSAT situation, it makes a request for a new job (`NEED_JOB` event). When the master has no more division of the search space to perform, it states the SAT problem is UNSAT.

Slaves. A slave may be in three different states: *idle*, *work*, and *work_ interrupt_ requested*. Initially, it is *idle* until it receives a `SOLVE` request from the master. Then, it moves to the *work* state and starts to process its assigned subspace. It may:

– find a solution, then emit a `SAT` event to the master, and move back to *idle*;
– end processing of the subspace, with an UNSAT result, then it emits a `NEED_JOB` event to the master, and move back to *idle*;
– receive an `INTERRUPT` event from the master, then, it moves to the *work_ interrupt_ requested* state and continues its processing until it reaches a stable state[10] according to the underlying sequential engine implementation. Then, it sends a `READY_TO_DIV` event to the master prior to move back to *idle*.

4.3 Implemented Heuristics

The divide-and-conquer component presented in the previous section should be generic enough to allow the implementation of any of the state-of-the-art strategies presented in Section 3. We selected some strategies to be implemented, keeping in mind that at least one of each family should be retained:

1. Techniques to Divide the Search Space (Section 3.1):
   we have implemented the guiding path method based on the use of assumptions. Since we want to be as generic as possible, we have not considered techniques adding constraints to the formula (because they require tagging mechanisms complex to implement to enable clause sharing).

[9] This policy may change dynamically over the execution of the solver.
[10] For example, in `Minisat`-based solvers, a stable state could correspond to the configuration of the solver after a restart.

2. Choosing Division Variables (Section 3.1):
   the different division heuristics we have implemented in the `MapleCOMSPS` solver[11], are: VSIDS, number of flips, and propagation rate.
3. Load Balancing (Section 3.2):
   a work-stealing mechanism was implemented to operate dynamic load balancing. The master selects targets using a FIFO policy (as in `Dolius`) moderated by a minimum computation time (2 seconds) for the workers in order to let these acquire a sufficient knowledge of the subspace.

The exchange of learnt clauses (Section 3.3) on any of the strategies we implemented is not restricted. This allows to reuse any of the already off-the-shelf strategies provided by the `Painless` framework.

Another important issue deals with the way new subspaces are allocated to workers. We provide two strategies:

– **Reuse:** the worker reuses the same object-solver all over its execution and the master feeds it with guiding paths;
– **Clone:** each time a new subspace is assigned to a worker, the master clones the object-solver from the target and provides the copy to the idle worker. Thus, the idle worker will benefit form the knowledge (VSIDS, locally learned clauses, etc.) of the target worker.

Hence, our `Painless`-based `D&C` component can thus be instantiated to produces solvers over six orthogonal axes: (1) technique to divide the search space; (2) technique to choose the division variables; (3) load balancing strategy; (4) the sharing strategy; (5) the subspace allocation technique; (6) and the used underlying sequential solver.

By lack of time for experimentations, we select for this paper 6 solvers: all based on `MapleCOMSPS`, and sharing all learnt clauses which LBD ≤ 4 (this value has been experimentally deduced). Table 1 summarizes the implemented `D&C` solvers we have used for our experiments in the next section.

	VSIDS	Number of flips	Propagation Rate
Reuse	`P-REUSE-VSIDS`	`P-REUSE-FLIPS`	`P-REUSE-PR`
Clone	`P-CLONE-VSIDS`	`P-CLONE-FLIPS`	`P-CLONE-PR`

Table 1: The `D&C` solvers we use for experiments in this paper

5 Evaluation

This section presents the results of experiments done with the six `D&C` solvers we presented in Section 4.3. We also did comparative experiments with state-of-art `D&C` solvers (`Treengeling` [6] and `MapleAmpharos` [26]).

[11] We used the version that won the main track of the SAT Competition in 2016 [19].

Treengeling is a cube-and-conquer solver based on the Lingeling sequential solver. MapleAmpharos is an adaptive divide-and-conquer based on the solver Ampharos [2], and using MapleCOMSPS as sequential solver. Comparing our new solvers with state-of-the-art ones (*e.g.,* not implemented on Painless) is a way to assess if our solution is competitive despite the genericity introduced by Painless and ad-hoc optimizations implemented in other solvers.

All experiments were executed on a multi-core machine with 12 physical cores (2 x Intel Xeon E5645 @ 2.40 GHz), and 64 GB of memory. Hyper-threading has been activated porting the number of logical cores to 24. We used the 400 instances of the parallel track of the SAT Competition 2018[12]. All experiments have been conducted using the following settings:

– each solver has been run once on each instance with a time-out of 5000 seconds (as in the SAT Competition);
– the number of used cores is limited to 23 (the remaining core is booked to the operating system);
– instances that were trivially solved by a solver (at the preprocessing phase) were removed, indeed in this case the D&C component of solvers is not enabled, these instances are then irrelevant for our case study.

Results of these experiences are summarized in Table 2. The different columns represent respectively: the total number of solved instances, the number of UNSAT solved instances, the number of SAT solved instances, and the PAR-2 score[13].

Table 2: Results of the different solvers

Solver	**ALL (360)**	**UNSAT**	**SAT**	**PAR-2**
P-CLONE-FLIPS	198	87	111	1732696.65
P-CLONE-PR	183	73	110	1871614.48
P-CLONE-VSIDS	183	77	106	1880281.54
P-REUSE-FLIPS	190	83	107	1796426.72
P-REUSE-PR	180	72	108	1938621.48
P-REUSE-VSIDS	184	75	109	1868619.43
MapleAmpharos	153	29	124	2190680.55
Treengeling	200	84	116	1810471.56

5.1 Comparing the Implemented Divide-and-Conquer Solvers

Figure 5 presents the cactus plot of the performances of the different D&C solvers. These differ in two orthogonal axes: the used subspace allocation technique, and the used division heuristic. We analyse here each axe separately.

[12] http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip
[13] The used measure in the annual SAT Competition.

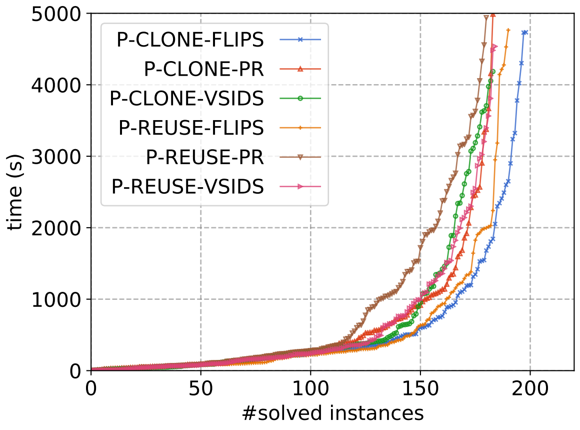


Fig. 5: Cactus plot of the different divide-and-conquer based solvers

When considering the allocation technique (clone *vs.* reuse), we can observe that the cloning based strategy is globally more efficient, even if it has a supplementary cost (due to the cloning phase). The scatter plots of Figure 6 confirm this observation (most plots are below the diagonal, showing evidence of a better average performance). We believe this is due to the local knowledge that is implicitly shared between the (cloned) workers.

When considering the division heuristics (VSIDS, number of flips, and propagation rate), we observe that number of flips based approach is better than the two others. Both, by the number of solved instances and the PAR-2 measure. This is particularly true when considering the cloning based strategy. VSIDS and propagation rate based solvers are almost identical.

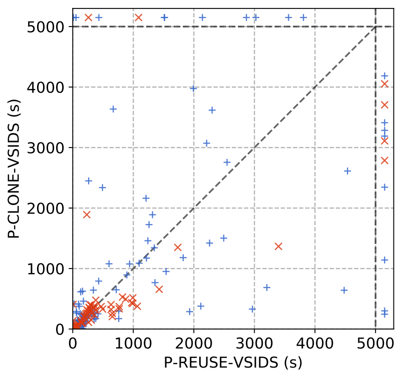
5.2 Comparison with State-of-the-Art Divide-and-Conquer

Figure 7 shows a cactus plot comparing our best divide-and-conquer (*i.e.,* `P-CLONE-FLIPS`) against `Treengeling` and `MapleAmpharos`.

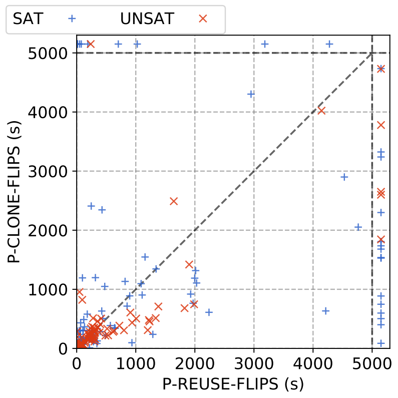
The `MapleAmpharos` solver seems to be less efficient than our tool, and solves less instances. When considering only the 123 instances that both solvers were able to solve, we can calculate the cumulative execution time of this intersection (CTI) for `MapleAmpharos` and `P-CLONE-FLIPS`: it is, respectively, 24h33min and 14h34min.

Although our tool solves 2 less instances as `Treengeling`, it has better PAR-2 measure. The CTI calculated on the 169 instances solved by both solvers, is 49h14min and 22h23min, respectively for `Treengeling` and `P-CLONE-FLIPS`. We can say that even if both solve almost the same number of instances, our `D&C` solver is faster. We clearly observe this phenomenon in Figure 7.

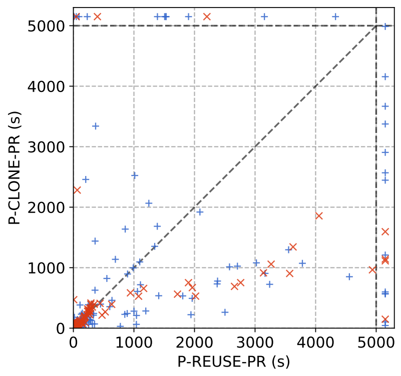
Thus, in addition to highlight the performance of our instantiation, this shows the effectiveness of the flip-based approach with respect to the well-proven cube-and-conquer strategies.



(a) VSIDS



(b) Number of flips



(c) Propagation rate

Fig. 6: Scatter plots of divide-and-conquer reusing *vs.* cloning solvers

6 Conclusion

This paper proposed an optimal implementation of several parallel SAT solvers using the divide-and-conquer (D&C) strategy that handle parallelisms by performing successive divisions of the search space.

Such an implementation was performed on top of the Painless framework that allows to easily deal with variants of strategies. Our Painless-based implementation can be customized and adapted over six orthogonal axes: (1) technique to divide the search space; (2) technique to choose the division variables; (3) load balancing strategy; (4) the sharing strategy; (5) the subspace allocation technique; (6) and the used underlying sequential solver.

This work shows that we have now a modular and efficient framework to explore new D&C strategies along these six axes. We were then able to make a fair comparison between numerous strategies.

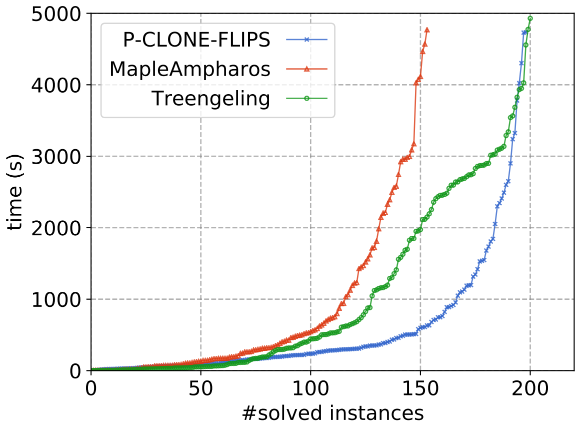


Fig. 7: Cactus plot of the best instantiated divide-and-conquer and state-of-the-art solvers

Among the numerous solvers we have available, we selected six of them for performance evaluation. Charts are provided to show how they competed, but also how they cope face to natively implemented `D&C` state-of-the-art solvers.

This study shows that the flip-based approach in association with the clone policy outperforms the other strategies whatever the used standard metrics is. Moreover, when compared with the state-of-the-art `D&C`-based solvers, our best solver shows to be very efficient and allows us to conclude the effectiveness of our modular platform based approach with respect to the well-competitive `D&C` solvers.

In the near future, we want to conduct more massive experiments to measure the impact of clauses sharing strategies in the `D&C` context, and evaluate the scalability of the various `D&C` approaches.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Piette, C.: Dolius: A distributed parallel SAT solving framework. In: Pragmatics of SAT International Workshop (POS) at SAT. pp. 1–11. Citeseer (2014)
2. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel SAT solver. In: Proceedings of the 22th International Conference of Principles and Practice of Constraint Programming (CP). pp. 30–48. Springer (2016)
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st International Joint Conferences on Artifical Intelligence (IJCAI). pp. 399–404. AAAI Press (2009)
4. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 197–205. Springer (2014)
5. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 156–172. Springer (2015)
6. Biere, A.: Cadical, lingeling, plingeling, treengeling and yalsat entering the SAT competition 2018. In: Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions. pp. 13–14. Department of Computer Science, University of Helsinki, Finland (2018)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 193–207. Springer (1999)
8. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press (2009)
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commununications of the ACM 5(7), 394–397 (1962)
10. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7(3), 201–215 (1960)
11. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6(4), 245–262 (2009)
12. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl SAT solvers by lookaheads. In: Proceedings of the 11th Haifa Verification Conference (HVC). pp. 50–65. Springer (2011)
13. Hyvärinen, A.E., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 430–435. Springer (2006)
14. Hyvärinen, A.E., Manthey, N.: Designing scalable parallel SAT solvers. In: Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 214–227. Springer (2012)
15. Jurkowiak, B., Li, C.M., Utard, G.: Parallelizing satz using dynamic workload balancing. Electronic Notes in Discrete Mathematics 9, 174–189 (2001)
16. Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence (ECAI). vol. 92, pp. 359–363 (1992)
17. Lanti, D., Manthey, N.: Sharing information in parallel search with search space partitioning. In: Proceedings of the 7th International Conference on Learning and Intelligent OptimizatioN (LION). pp. 52–58. Springer (2013)

18. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: a framework for parallel SAT solving. In: Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 233–250. Springer (2017)
19. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maplecomsps, maplecomsps lrb, maplecomsps chb. In: Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions. p. 52. Department of Computer Science, University of Helsinki, Finland (2016)
20. Lynce, I., Marques-Silva, J.: SAT in bioinformatics: Making the case with haplotype inference. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 136–141. Springer (2006)
21. Marques-Silva, J.P., Sakallah, K.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
22. Martins, R., Manquinho, V., Lynce, I.: Improving search space splitting for parallel SAT solving. In: Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI). vol. 1, pp. 336–343. IEEE (2010)
23. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning 24(1), 165–203 (2000)
24. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC). pp. 267–275. ACM (1996)
25. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC). pp. 530–535. ACM (2001)
26. Nejati, S., Newsham, Z., Scott, J., Liang, J.H., Gebotys, C., Poupart, P., Ganesh, V.: A propagation rate based splitting heuristic for divide-and-conquer solvers. In: Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 251–260. Springer (2017)
27. Plaza, S., Markov, I., Bertacco, V.: Low-latency SAT solving on multicore processors with priority scheduling and xor partitioning. In: the 17th International Workshop on Logic and Synthesis (IWLS) at DAC (2008)
28. Silva, J.P.M., Sakallah, K.A.: Grasp—a new search algorithm for satisfiability. In: Proceedings of the 16th IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 220–227. IEEE (1997)
29. 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)
30. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 20th IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 279–285. IEEE (2001)