



On the Usefulness of Clause Strengthening in Parallel SAT Solving

Vincent Vallade¹(✉), Ludovic Le Frioux², Souheib Baair^{1,3}, Julien Sopena^{1,4},
and Fabrice Kordon¹

¹ Sorbonne Université, CNRS, LIP6, UMR 7606, Paris, France
`vincent.vallade@lip6.fr`

² LRDE, EPITA, Le Kremlin-Bicêtre, France

³ Université Paris Nanterre, Nanterre, France

⁴ Inria, DELYS Team, Paris, France

Abstract. In the context of parallel SATisfiability solving, this paper presents an implementation and evaluation of a clause strengthening algorithm. The developed component can be easily combined with (virtually) any CDCL-like SAT solver. Our implementation is integrated as a part of Painless, a generic and modular framework for building parallel SAT solvers.

Keywords: Parallel satisfiability · Tool · Strengthening · Clause sharing · Portfolio · Divide-and-conquer

1 Introduction

Modern CDCL SAT solvers [1, 12] have been successfully used to solve a wide variety of real-world problems, such as those issued from hardware and software verification [4].

With the omnipresence of many-core machines, these solvers have been adapted to become parallel [3]. In this context, a key feature in the efficiency is information sharing. This is usually implemented as sets of new (learnt) lemmas that are exchanged between the different participants of the parallelization solving strategy (i.e., the underlying sequential solvers).

Besides, it is well admitted that the shorter the learnt lemmas the more powerful they are. This explained the proposal of different techniques based on resolution to shorten them [7, 8, 15, 16]. A process known as *strengthening*. Potentially as difficult as the SAT problem itself, the strengthening of those learnt lemmas can benefit from parallelization [17].

This paper presents the implementation and evaluation of a parallel strengthening algorithm inspired from [17]. Our implementation is integrated as a part of Painless [9], a framework for building parallel SAT solvers.

Paper Structure. Section 2 introduces some background. Strengthening is presented in Sect. 3. Its implementation is described in Sect. 4. Some experimental results are depicted in Sect. 5. Section 6 concludes the paper.

2 Background

This section introduces useful background used in the remaining of this paper.

Boolean Satisfiability. A *propositional variable* is a variable that has two possible values: *true* or *false*. A *literal* is a propositional variable or its negation (NOT). A *clause* is a finite disjunction (OR) of literals. A clause with a unique literal is called *unit clause*. A *conjunctive normal form (CNF) formula* is a finite conjunction (AND) of clauses. In the rest of the paper clauses are represented by the set of their literals, and formulas by the set of their clauses. Let F be a formula, an *assignment* of variables of F is defined as a function $\mathcal{A} : \mathcal{V} \rightarrow \{\text{true}, \text{false}\}$, where \mathcal{V} is the set of variables of F . A clause is satisfied when at least one of its literals is evaluated to *true*. A formula is satisfied if all its clauses are evaluated to *true*. A formula is said to be SAT if there is at least one assignment of its variables that makes it *true*; it is reported to be UNSAT otherwise. The Boolean satisfiability (SAT) problem consists in determining if a given formula is SAT or UNSAT.

CDCL Algorithm. Conflict-driven clause learning algorithm [14, 18] is used in almost all (complete) modern SAT solvers. It enumerates assignments for the given formula. Variables' values are forced using *unit propagation* [5] (*i.e.*, fixing recursively the values of variables in unit clauses). If an empty clause is generated a conflict has been reached. The reasons are studied and a learnt clause is derived and stored. The search backtracks and starts over. If unit propagation does not generate a conflict, a guess is done (branching) to grow up the current assignment. The search ends if a satisfying assignment has been found or if all have been checked without finding solutions.

Let F be a formula, unit propagation can be iteratively applied for a given partial assignment \mathcal{A} : `iterativeUnitPropagation(F, \mathcal{A})` produces the set of assignments implied by this operation. $F|_{\mathcal{A}}$ returns the formula simplified by the iterative unit propagations of \mathcal{A} on F .

3 Strengthening Algorithm

The pseudo-code of the strengthening algorithm we implemented in our tool is presented in Algorithm 1. The theoretical basics of this technique are presented in [17]. This section only focuses on the technical details.

Algorithm 1 takes a clause C_{in} as input, and, potentially outputs a reduced size (strengthened) clause, w.r.t. C_{in} , (lines 8 and 14). It considers an empty assignment \mathcal{A} (line 3), the knowledge of all the clauses of the problem F , and it manages its own set of learnt clauses L_R (line 4) empty at the beginning of the program.

To achieve its strengthening task, Algorithm 1 iteratively assigns a false value to each literal of the clause C_{in} , until it reaches a conflict or it assigns successfully all literals of the input clause. Therefore, there are two possible outputs, respectively C_{new} and C_{out} .

Algorithm 1: Strengthening algorithm

```

1 function strengthen( $C_{in}$ : clause) : the strengthened clause
2    $C_{out} := \emptyset$ 
3    $\mathcal{A} := \emptyset$ 
4    $F' := F \cup L_R$ 
5   for  $l \in (C_{in} \setminus C_{out})$  s.t.  $\neg l \notin \text{iterativeUnitPropagation}(F', \mathcal{A})$  do
6     if  $\emptyset \in F'|_{\mathcal{A}}$  then
7        $(L_R, C_{new}) := \text{analyze}(F', \mathcal{A})$ 
8       return  $C_{new}$ 
9     end
10     $C_{out} = C_{out} \cup \{l\}$ 
11     $\mathcal{A} = \mathcal{A} \cup \{\neg l\}$ 
12  end
13   $L_R = L_R \cup \{C_{out}\}$ 
14  return  $C_{out}$ 

```

At each iteration, Algorithm 1 picks a literal whose complementary is not implied by the current assignment ($\neg l \notin \text{iterativeUnitPropagation}(F', \mathcal{A})$). This ensures the stripping of the input clause from all literals that are implied by the rest of the clause. Then, it executes a unit propagation (line 5). If no conflict is discovered, the literal is added to the output clause C_{out} and its negation is added to the set of assignment \mathcal{A} (line 10–11).

When a conflict is reached (line 6), function `analyze()` is then called (line 7): it executes a sequence of backtracking, unit propagation, and conflict analysis until getting out of the conflict or emptying the set \mathcal{A} . During this phase, the algorithm learns new clauses (that are added to L_R). When `analyze()` reaches a zone without conflict (while assuming \mathcal{A}), it generates the clause C_{new} that is returned (line 8). This last is composed of the set of literals: $\{l | \neg l \in \mathcal{A}\} \cup \{k\}$, k being some literal of $C_{out} \notin \mathcal{A}$.

If all the literal of C_{in} are assigned successfully, then the clause C_{out} is added to L_R and then returned (lines 13–14).

4 Implementation

Our implementation is based on Painless [9] which is a framework allowing the implementation of parallel SAT solvers for many-core environments. The main components of Painless are: working organization, clause sharing, and sequential engines. For this work, we focused on the third component and implemented a reducer solver that can be included in all Painless' configurations.¹

About the Painless Framework. The main idea of the framework is to separate the technical components (*e.g.*, dedicated to concurrent programming

¹ This version of Painless can be found at <https://github.com/lip6/painless>, branch strengthening.

aspects) from those implementing heuristics and optimizations embedded in a parallel SAT solver. Three main components arise when treating parallel SAT solvers: *Sequential Engines*, *Parallelisation*, and *Sharing*. These form the global architecture of Painless. They can be instantiated independently to produce new complete solvers.

The core element considered here is a sequential SAT solver. This can be any CDCL-like solver. Technically, these engines are operated through a generic *SolverInterface* providing basics of sequential solvers: *solve*, *add clauses*, etc.

To build a parallel solver using the aforementioned engines, one needs a parallelisation strategy (e.g., *s* portfolio, divide-and-conquer). In Painless, a strategy is represented by a tree-structure of arbitrary depth. The internal nodes of the tree (*WorkingStrategy*) represent parallelisation strategies, and leaves are core engines operated by a thread (*SequentialWorker*).

In Painless, solvers can export (import) clauses to (from) the others during the resolution process. The sharing of these learnt clauses is dedicated to particular components called *Sharers*. Each *Sharer* is in charge of sets of producers and consumers and its behaviour reduces to a loop of sleeping and exchange phases *w.r.t.* to a given *SharingStrategy*.

Implementing Strengthening. This section presents the implementation of strengthening we included into the Painless framework. The development of such a component has been designed to be easily used in combination with all other mechanisms provided by Painless.

The *Reducer* engine of Fig. 1 implements Algorithm 1. As it can be easily observed, the main component of this algorithm are iterative unit propagation and analysis (based on assumptions) procedures. These are also the usual components provided by any CDCL-like SAT solver.

Therefore, we implemented the strengthening operation as a decorator of *SolverInterface*. This decorator is a *SolverInterface* itself that uses, by delegation, another *SolverInterface* to apply the strengthening (see Fig. 1).

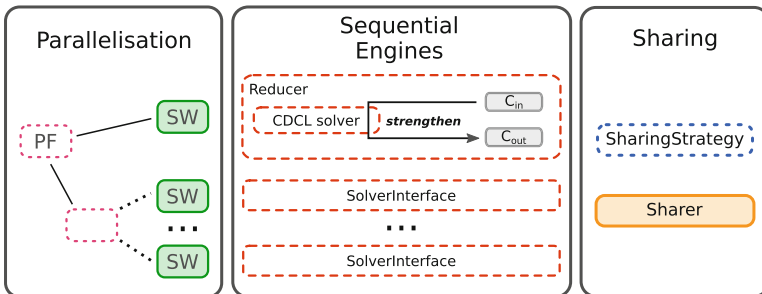


Fig. 1. Parallel strengthening architecture

The CDCL solver needs to be able to solve a formula with a set of assumptions. Assumptions are literals with a predefined value that the solver must

accept as immutable. This is how we implemented the loop in Algorithm 1. We give the negation of the learnt clause as assumptions to the solver, which stops the resolution when a conflict is reached or when the solver has branched on all the assumptions. The solver must also be able to express a conflict only in terms of assumptions, i.e. the set of literals returned by the analysis contains only literals present in the initial set of assumptions.

The *Reducer* is always at the root of a portfolio. For example, if one wants to implement a divide-and-conquer solver complemented by a *Reducer*, they must create a portfolio with a *Reducer* and a divide-and-conquer as workers. This is extremely easy to do thanks to the composite nature of Painless' *Parallelisation* engine. The *Reducer* is both a consumer and a producer of the Sharer. It receives clauses, strengthened them and shares them back after.

5 Empirical Study

To assess the performances of the developed component and study its impact in different parallel solvers, we integrated our *Reducer* in several parallelisation strategies. We then conducted a set of experiments to compare the results.

Solvers Description. All parallel solvers we constructed, but one, are based on P-MCOMSPS [11]. It implements a portfolio strategy [6] (PF) and uses MapleCOMSPS [13] as sequential engine. The solvers differ however by their sharing strategies.

One of the main heuristics used in sharing strategies is the so-called Literal Block Distance (LBD) measure: the LBD of a clause is the number of decision levels represented in that clause. It is fairly admitted that the lower the LBD, the better the clause [1]. In a parallel context, it is useful to share these low LBD clauses.

We therefore derived the following strategies: **AI**, only learnt clauses with an LBD value less or equal than a threshold are shared. This threshold is additionally increased if not enough clauses are exchanged [2]; **Li** shares only learnt clauses with an LBD value $\leq i$. Hence, we ended up by developing the solver P-MCOMSPS-AI², the solver P-MCOMSPS-L2³ and the solver P-MCOMSPS-L4 (*L4* is a new untested yet strategy).

To complete the picture, we also developed a divide-and-conquer (DC) solver that uses *L4* sharing strategy. We call this solver DC-MCOMSPS-L4 [10].

For each of these solvers, we created its counterpart including the *Reducer* component. We called them by extending their names by **-REDUCE** (e.g., P-MCOMSPS-L4-REDUCE). It is important to note that we do not use an additional core for the *Reducer*, e.g., if we use 12 cores for P-MCOMSPS-L4, we also use 12 cores for P-MCOMSPS-L4-REDUCE, one thread performs the strengthening instead of the CDCL algorithm.

² *AI* is the strategy used by the winner of the parallel track of 2018 SAT competition.

³ *L2* is the strategy used by the second of the parallel track of 2018 SAT competition.

Table 1. Results of the different solvers on the SAT benchmark 2018

Parallelisation	Solvers	PAR-2	CTI	UNSAT	SAT	SCR(400)
PF	P-MCOMSPS-L4	363h06	26h53	115	165	280
	P-MCOMSPS-L4-REDUCE	342h33	21h47	121	168	289
	P-MCOMSPS-L2	379h32	23h04	108	165	273
	P-MCOMSPS-L2-REDUCE	371h53	20h45	115	163	278
	P-MCOMSPS-AI	356h13	37h10	121	165	286
	P-MCOMSPS-AI-REDUCE	342h36	32h15	125	167	292
DC	DC-MCOMSPS-L4	448h34	17h17	100	146	246
	DC-MCOMSPS-L4-REDUCE	437h44	18h59	103	149	252

Experimental Results. For the evaluation we use the main benchmark of the SAT competition 2018⁴ which contains 400 instances. All jobs were run on an Intel Xeon CPUs @ 2.40 GHz and 1.48 TB of RAM. Solvers have been launched with 12 threads, a 150 GB memory limit, and a 5000s timeout (the timeout is the same as for the SAT competitions).

The performance of our solvers is evaluated using the following success metrics: penalized average runtime (PAR-2) sums the execution time of a solver and penalizes the executions that exceed the timeout with a factor 2; solution-count ranking (SCR) counts the number of problems solved by a solver; cumulative time of the intersection (CTI) sums the execution time of a solver on the problems solved by all the solvers.

Table 1 presents the results of our experiments, where each solver is compared to its counterpart (with a *Reducer* component). The shaded cells indicate which one of the two solvers has the best results. We observe that in all metrics, but two cases, the versions with a *Reducer* are better: more instances are solved and better PAR-2 values are obtained in all cases. Only CTI of the DC version is not as good as the other values. Also, the gains in the number of instances solved appears to be greater in the UNSAT category, but the number of SAT instances also improves.

To go further in our evaluation, we measured the minimisation capabilities of the *Reducer* on instances that each solver could actually solve, while discarding those where the *Reducer* did not receive any clause (problem solved too quickly): (1) P-MCOMSPS-L4-REDUCE (255 instances), 44.21% of the clauses treated by the *Reducer* are actually shortened. The mean size of these clauses after strengthening is 25.45% less than the mean of their original size; (2) P-MCOMSPS-L2-REDUCE (257 instances), treated 32.59% of the clauses and it lower their size by 23.67%; (3) P-MCOMSPS-AI-REDUCE (258 instances) treated 34.79% clauses and reduced by 27.75%; (4) DC-MCOMSPS-L4-REDUCE (245 instances) reduced 28.80% clauses by 18.86%. In conclusion, the *Reducer* succeeded to reduce 1/3 of the clauses it receives by 1/4 of their size.

⁴ <http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip>.

6 Conclusion

This paper presents an implementation of clause strengthening [17] which has been integrated into Painless [9]. Thanks to the modularity of Painless, we were able to test the efficiency of strengthening within different configurations of parallel SAT solvers.

In this study, we used several sharing strategies and different parallelisation paradigms (*i.e.*, portfolio and divide-and-conquer). Our experiments show that having a core dedicated to strengthening improves the performance of our parallel solvers whatever the configuration is (including the winner configuration from the SAT competition 2018).

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Joint Conferences on Artificial Intelligence (IJCAI), pp. 399–404. AAAI Press (2009)
2. Balyo, T., Sanders, P., Sinz, C.: HordeSat: a massively parallel portfolio SAT solver. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 156–172. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_12
3. Balyo, T., Sinz, C.: Parallel satisfiability. Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63516-3_1
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
6. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)
7. Han, H., Somenzi, F.: Alembic: an efficient algorithm for CNF preprocessing. In: Proceedings of the 44th Annual Design Automation Conference. DAC 2007, pp. 582–587. Association for Computing Machinery, New York (2007). <https://doi.org/10.1145/1278480.1278628>
8. Heule, M.J.H., Jarvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 201–215. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_17
9. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: a framework for parallel SAT solving. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 233–250. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_15
10. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 135–151. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_8
11. Le Frioux, L., Metin, H., Baarir, S., Colange, M., Sopena, J., Kordon, F.: painless-mcomsps and painless-mcomsps-sym. In: Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, pp. 33–34. Department of Computer Science, University of Helsinki, Finland (2018)

12. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
13. 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)
14. Marques-Silva, J.P., Sakallah, K.: GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* **48**(5), 506–521 (1999)
15. Piette, C., Hamadi, Y., Saïs, L.: Vivifying propositional clausal formulae. In: Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence, pp. 525–529. IOS Press, NLD (2008)
16. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 237–243. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_23
17. Wieringa, S., Heljanko, K.: Concurrent clause strengthening. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 116–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_10
18. 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)