Lazy clause exchange policy for parallel SAT solvers

Gilles Audemard[1]⋆ and Laurent Simon[2]⋆⋆

[1] Univ. Lille-Nord de France. CRIL/CNRS UMR 8188, Lens.
[2] Univ. Bordeaux. LABRI, Bordeaux

Abstract. Managing learnt clauses among a parallel, memory shared, SAT solver is a crucial but difficult task. Based on some statistical experiments made on learnt clauses, we propose a simple parallel version of Glucose that uses a lazy policy to exchange clauses between cores. This policy does not send a clause when it is learnt, but later, when it has a chance to be useful locally. We also propose a strategy for clauses importation that put them in "probation" before a potential entry in the search, thus limiting the negative impact of high importation rates, both in terms of noise and decreasing propagation speed.

1 Introduction

The success story of SAT solving is one of the most impressive in recent computer science history. The theoretical and practical progresses observed in the area had a direct impact in a number of connected areas. SAT solvers are nowadays used in many critical applications (BMC [5], Bio-informatics [17] ...) by direct encodings of problems to propositional logic (often leading to huge formulas), or by using SAT solvers on an abstraction level only.

However, if until now measured progresses are quite impressive, the recent trends in computer architecture are forcing the community to study new efficient frameworks, by considering the native parallel (and sometimes massively parallel) architecture of current and upcoming computers. CPU speed is stalling, but the number of cores is increasing. Computers with one shared memory and a large number of cores are the norm today. A few specialized cards even allow more than two hundreds threads on the same board. Thus, designing efficient and scalable parallel SAT solvers is now a crucial challenge for the community [11]. Existing approaches can be roughly partitioned in two. Firstly, the "portfolio" approach tries to launch in parallel a set of solvers on the same formula. This can be trivially done by running the best known solvers without any communications [19] or, more interestingly, with communication between threads. This communication is generally limited to learnt clauses sharing [10, 1, 6]. Secondly, the divide and conquer approach tries first to reduce the whole formula in smaller ones and then solve them [2, 13, 12] with or without any communications. Note that some attempts have been made on combination of portfolio and divide and conquer approaches [7].

In our approach, we would like to consider CDCL solvers as clauses producer engines and thus, in this case, the current divide and conquer paradigm may not be ideal

⋆ This work has been supported by CNRS and OSEO, under the ISI project "Pajero".
⋆⋆ This work was possible thanks to the BIRS meeting 14w5101.

because the division is made on the variable search space, not on the proof space. When designing a clause sharing parallel CDCL solver, an important question arises: which clauses to export and import? The more is not the best. Importing too many clauses will completely paralyze the considered thread. If we have N threads sending its clause with probability p then, after C conflicts, each thread will have on average C learnt clauses and $p \times (N-1) \times C$ imported clauses. Thus, keeping p as low as possible is critical. Too many imported clauses by too many distinct threads will destroy the effort of the current thread to focus on a subproblem. This problem is not new and was already mentioned and partially answered by one of the first parallel SAT solver: MANYSAT [10, 8].

Let us summarize the original contributions of our approach. First, we try to carefully identify clauses that have a chance to be useful locally (even locally, just a part of produced clauses are in fact really useful). These clauses will be detected "lazily" before exporting them. Secondly, we don't directly import clauses. We put them in "probation" before adding them to the clauses database, thus limiting their impact on the current search. In the following section, we review the different approaches proposed for parallelizing Modern SAT solvers. Then, in section 3, we detail the principal "Lazy" clause exchange policy proposed in GLUCOSE-SYRUP, our parallel version of GLUCOSE. Then, we propose some experiments (section 5) and conclude.

2 Preliminaries and previous works

We assume the reader familiar with the essentials of propositional logic, SAT solving and CDCL solvers. These solvers are branching on literals and, at any step of the search, ensure that all the unit clauses w.r.t the current partial assignment are correctly "propagated" until an empty clause is found (a backtrack is then fired, or the unsatisfiability is proven) or a total assignment is reached (the formula is SAT). Each time a conflict occurs, a clause (called *asserting clause*) is learnt and is used to force backtracking, leading to new propagations. Of course, many additional ingredients are essential but reviewing all of them will clearly be beyond the scope of this paper.

The important point to be emphasized here, even for the non specialist, is that solvers are learning a lot of clauses (more than 5000 per second), partially guided in its search by previous learnt clauses forcing new unit propagations. The management of the clauses database was firstly pointed out as an essential ingredient with the design of GLUCOSE [4]. Indeed, keeping too many learnt clauses will slow down the unit propagation process, while deleting too many of them will break the overall learning benefit. Consequently, identifying good learnt clauses – relevant to the (future) proof derivation – is clearly an important challenge. The first proposed quality measure followed the success of the activity-based "VSIDS" heuristic [18]. More precisely, a learnt clause is considered relevant (in the future) to the proof, if it is involved more often in recent conflicts, *i.e.* used to derive asserting clauses by resolution. This deletion strategy supposes that a useful clause in the past would be useful in the future. In [4], the authors proposed a more accurate measure called LBD (*Literal Block Distance*) to estimate the quality of a learnt clause. This measure is based on the number of distinct decision levels occurring in a learnt clause and computed when the clause is learnt. Intensive

experiments demonstrated that clauses with small LBD values are used more often than those of higher LBD ones. This measure is important here because it seems to offer a good prediction for the quality of a clause, and it seems to be a good starting point if we may want to build a good clause exchange policy between threads. However, as we will see, the LBD measure seems essentially relevant to the current search only. Even if it has been proven to be more relevant than size (see below the PLINGELING strategy), a small LBD clause may not be of great interest for another thread.

2.1 About Clause-Sharing Parallel Approaches

A number of previous works have been proposed around Portfolio approaches, with more or less cooperation/diversification between threads. The main idea is to exploit the complementarity between different sequential CDCL strategies to let them compete on the same formula with more or less cooperation between them [10, 6, 16, 1]. Each thread deals with the whole formula and cooperation is achieved through the exchange of learnt clauses. Non-Portfolio approaches are mostly based on the divide-and-conquer paradigm [2, 13]. We here focus on Portfolio parallel approaches in this section.

As mentioned in the introduction, the size of the learnt clause database is crucial for sequential solvers. This is not only true for maintaining a good unit propagation speed, but this seems also essential to guide the solver to the best possible proof it can build. These observations are even more crucial when many threads are cooperating. For a parallel portfolio SAT solver, it is not desirable to share as many clauses as possible and, obviously, each of the clause-sharing portfolio approaches mentioned above had to develop its own strategy to carefully select the clauses to share. A first and quite natural solution to limit the number of exported clauses is to simply share the smallest clauses according to their size. This was the strategy adopted in [10]. Based on the observation that small clauses appear less and less during the search, authors of MANYSAT proposed a very nice dynamic clause sharing policy using pairwise size limits to control the exchange between threads [9]. In the same paper, they also anecdotally proposed another dynamic policy based on the activity of variables according to the VSIDS heuristic. However, such weighting function is highly fickle and, unfortunately, the top-ranked variable (according to VSIDS) may not be of any interest only 0.1s later (according to the same VSIDS). It is thus hopeless to try to directly rely on this highly dynamic strategy to measure the quality of imported clauses. In `Penelope` [1], authors use the freezing strategy to manage learnt clauses [3] allowing to share much more clauses without a high overhead. Finally, PLINGELING shares all clauses with a size less than 40 and LBD less than 8 [6].

2.2 Portfolio or not Portfolio

We chose to focus, in this paper, on a Clause-Sharing Parallel Approach of GLUCOSE engines. This approach is often misleadingly called "Portfolio" because, in general, each engine must have its own configuration to ensure orthogonal searches between threads. However, considering that GLUCOSE is seen as an efficient proof producer, this idea of "orthogonal" search on variables assignments is not clear. We would like to keep the word "portfolio" only to approaches that tries to take advantage of running many

distinct solvers, each of them specialized on a subset of problems, and thus focusing on the competition between each configuration strengths. In our approach, we will restrict the "orthogonal" search to its minimum. Thus, we rather see our approach as a simple parallelization effort of the same engine rather than a "portfolio" one. Our final goal is to see all solvers working together to produce a single proof, as short as possible.

3 Lazy clause exchange

This section describes the strategy we propose for parallelizing GLUCOSE. This strategy is only based on how to identify "good" clauses to export and to import. The parallel solver is called "GLUCOSE-SYRUP" (i.e. *a lot of glucose*).

3.1 Identifying useless clauses?

The identification of useless clauses is still an open question for sequential solvers. We thus do not pretend to fully answer it in this section. However, let us take some time in this section to report a set of experiments on sequential solvers that motivated the strategies proposed in GLUCOSE-SYRUP.

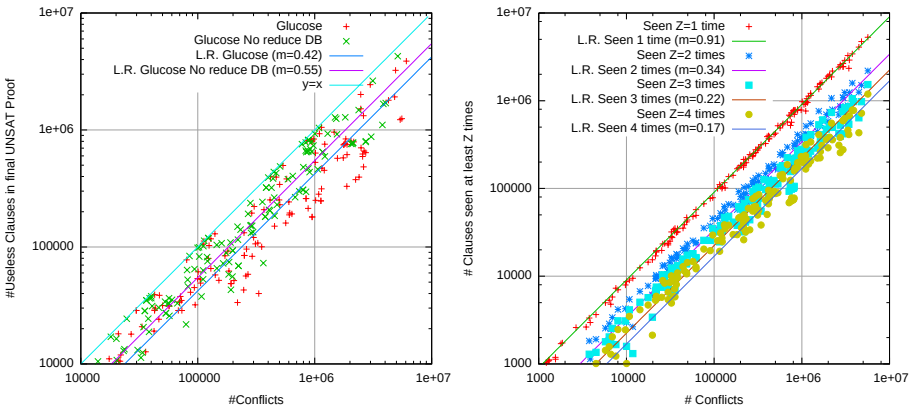


Fig. 1. (Left) Useless clauses in the final UNSAT proof w.r.t the total number of generated clauses. Glucose no reduce DB is a hacked version of GLUCOSE that do not perform any learnt clause removal. Experiments are done on a set of 250 UNSAT problems from competitions 2011 and 2013. Only successful run are collected. "L.R." stands for "Linear Regression" with $y = m \times x + n$. **(Right)** Scatter plot of the number of conflicts (X axis) against the number of clauses seen at least Z=$\{1, 2, 3, 4\}$ times for all the successful launches (on SAT 2011, satelited, problems)

Viewing CDCL solvers as clauses producers is not the mainstream approach. However, as it was reported in [15], this may be one of the key points for an efficient parallelization. This work suggested the following very simple experiment. When an instance

is UNSAT (proven by GLUCOSE), we identify clauses that occur in the proof, and measure how many of them are *useless* for the proof. The term "useless" is now clearly defined: it does not occur in the final proof for UNSAT. However, it should be noticed here that this definition can be misleading. A "useless" clause may be crucial at some stage of the search to update the heuristic or to propagate a literal earlier in the search tree. Thus, this notion should be used with caution.

This being said, Figure 1-Left shows a surprisingly large number of useless clauses. On the original GLUCOSE solver, 45% of the learnt clauses are, on average, not useful. This result is not the only surprise here. When GLUCOSE keeps all its clauses (called "Glucose No Reduce DB" on the figure), this number is even more important (55%!). This mean that keeping all the clauses, even if the final number of conflicts is smaller than the original GLUCOSE, leads to a larger proportion of useless clauses. We could have expected the opposite to happen: the aggressive clause database reduction in GLUCOSE throws away many clauses. Those clauses will not have a chance to occur in the proof afterwards. As a very short conclusion on this figure, we see that even for a single engine, considering the usefulness of a clause is not an easy task. Moreover, around half of the generated clauses are not useful. Sending them to other threads may not be the right move to do.

3.2 Lazy Exportation of Interesting Clauses

Before presenting the export strategy in GLUCOSE-SYRUP, let us focus now on Figure 1-Right. This figure shows how many clauses are seen at least Z=$\{1,2,3,4\}$ times during all conflicts analysis of GLUCOSE (not propagations). The figure already shows that 91% of the clauses, on all the problems, are seen at least once. However, when Z=2, this ratio drops to 34%, then 22% and 17% for Z=4. This suggests to export only clauses seen at least Z times, and to fix Z>1. We propose to simply fix Z=2 to already get rid of 66% (100-34%) of the locally generated clauses, and to send only clauses seen two times. The strategy is called "*lazy*" because we do not try to guess in advance the usefulness of a clause. Instead, we simply (and somehow "lazily") wait for the clause to be seen twice in the conflict analysis.

To refine the above observations, we conducted two more experiments. Firstly, we supposed that, in many cases, a learnt clause had a high probability to be seen in the very next conflict analysis, because the clause is immediately propagated and is clearly at one of the deepest levels of the search tree. However, as clearly shown Figure 2-Left, only 41% of the learnt clauses are immediately used for conflict analysis (remember that 91% of the clauses are seen at least once). This means that we really need a lazy strategy to identify interesting clauses: we cannot suppose any locality (in the number of conflicts) for identifying when a clause will be used for the second time.

As we already pointed out in the second section, PLINGELING is using a fixed strategy to filter the exported clauses. It restricts the exportations to clauses of LBD smaller than 8 and size smaller than 40. We use a more flexible limit in our approach. We automatically adapt the LBD and size thresholds according to the characteristics of the current clauses in the learnt clause database. Each time a clause database reduction is fired, the current median LBD value and the average size of learnt clauses are updated.

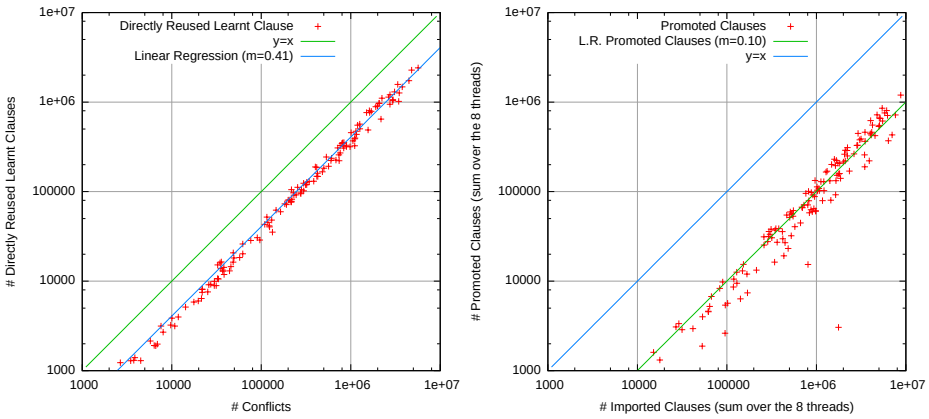


Fig. 2. Left Scatter plot of the number of conflicts (X axis) against the number of clauses directly reused in the next conflict analysis (on a single engine GLUCOSE). **Right** Scatter plot of the number of imported clauses (1-Watched clauses) against promoted clauses (clauses found empty, and pushed to the 2-Watched literal scheme) on successful run of GLUCOSE-SYRUP with 8 threads on the SAT'11 competition benchmarks (after SatElite)

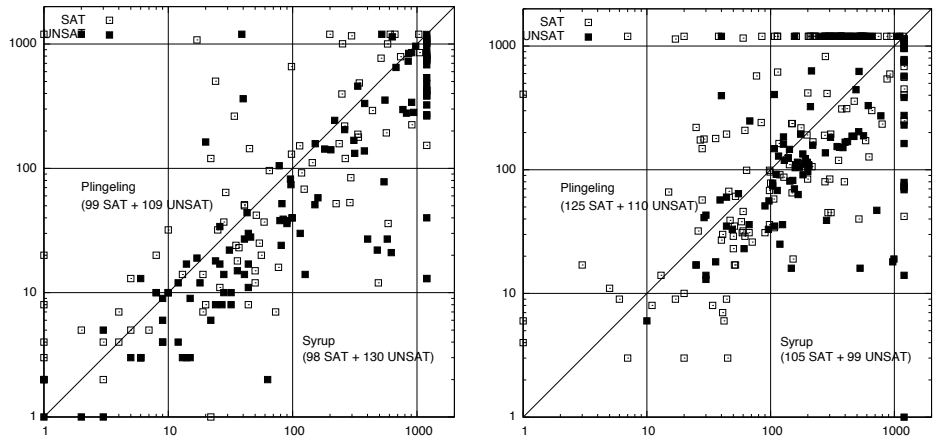
In most of the cases, we observed that the dynamic values are more relaxed than the fixed values of PLINGELING.

Of course, unit clauses and binary clauses are exported without any restriction, as soon as they are learnt. We extended this strategy to all glue clauses (clauses of LBD=2). Thus the above strategy is for clauses of at least size 3 that are not glues.

3.3 Lazy importation of clauses

We have seen how carefully choosing which clauses to send can be crucial for limiting the communication overhead. The extra work for importing clauses can also be limited if we can consider the following four points. (1) Importing clauses after each conflict may have an important impact (and a negative one) on all solvers strategies (need to check the trail, and to backjump when necessary, breaking the solver locality, ...) ; (2) Importing clauses may have an important impact on the cleaning strategies of GLUCOSE (the learnt clause database will increase much more faster) ; (3) The more is not the best. A "*bad*" clause may force the solver to propagate a literal in the wrong direction, i.e. it will not be able to properly explore the current subproblem ; (4) There is no simple way of computing the LBD of an incoming clause. This measure cannot be used because LBD is relative to the current search of each solver, and a good clause for one thread may not be good for another one. So, before adding a clause to the solver database, one has to carefully ensure that the clause is useful for the current search.

For the first point (1), we decided to import clauses (unary, binary and others) only when the solver is at decision level 0, *i.e.* right after a restart or right after it learnt a unit clause. This event occurs a lot during the search (many times per second) and this



(a) SAT'11 competition (b) SAT'13 competition

Fig. 3. GLUCOSE-SYRUP VS PLINGELING on SAT competitions benchmarks, application track. Each dot represents an instance. A dot below the diagonal indicates an instance solved faster with GLUCOSE-SYRUP. On SAT'11 Problems (a): PLINGELING: 208 / GLUCOSE-SYRUP: 228. On SAT'13 Problems(b): PLINGELING: 235 / GLUCOSE-SYRUP: 204

rate is clearly sufficient for a good collaboration. For the second point, (2) we import clauses in another set of clauses, that can have its own cleaning rules and will not pollute GLUCOSE cleaning strategies on its learnt clauses.

The main "Lazy" solution we propose is a solution for the two last items, (3) and (4). It is somehow related to the PSM strategy [3]: when a clause is imported, we watch it only by one literal. This watching scheme does not guarantee anymore that all unit propagations are performed after each decision. However, this is sufficient to ensure that any conflicting clause will be detected during unit propagation. The interest of this technique is twofold. Firstly, the clause will not pollute the current search of the solver, except when it is falsified. Secondly, the cost for handling the set of imported clause is heavily reduced. This technique can be viewed as a more reactive PSM strategy.

Figure 2-Right shows that only 10% of the imported clauses are falsified at some point. It demonstrates how well founded is our strategy for the importation: 90% of the clauses are never falsified by the solver strategy. Technically speaking, as soon as an imported clause is falsified, it is "promoted", i.e. we watch it with 2 literals like an internal learnt clause. The clause is then part of the solver search strategy, and can be used for propagation.

4 Experiments

In this section, we compare PLINGELING [6], the winner of the SAT'13 competition with our first version of GLUCOSE-SYRUP. Let us notice here that this version is still preliminary in the sense that absolutely no tuning has been conducted on the set of pa-

rameters we chose for the 8 threads. All threads are identical except their parameter playing on the VSIDS scoring scheme. One thread is however configured like GLUCOSE 2.0 version.

Figure 3 compares PLINGELING [6], the winner of the SAT'13 competition against GLUCOSE-SYRUP using the classical scatter plots. We use two test sets of problems, because each solver has its own strengths and weaknesses. Let us start with the SAT'11 competition, application track, problems. For this test set, GLUCOSE-SYRUP is able to solve 228 instances (98 SAT and 130 UNSAT) whereas PLINGELING is able to solve 208 instances (99 SAT and 109 UNSAT). Note that the sequential version of GLUCOSE only solves 187 instances (87 SAT and 100 UNSAT). GLUCOSE-SYRUP clearly extends the efficiency of GLUCOSE on UNSAT problems to the parallel case. Moreover, many points are below the diagonal indicating that, in many cases, our solver is faster (even for SAT instances) than PLINGELING. This result may be partially explained by all the inprocessing [14] techniques embedded in PLINGELING, that may not be efficient enough with the time we fixed. More importantly, we think the big gap observed for UNSAT instances can come from the ability of GLUCOSE-SYRUP to share and exploit promising clauses. It is also fair to notice that GLUCOSE, the underlying sequential engine is quite good in solving those UNSAT instances. Let us continue with instances coming from SAT'13 competition. Here, the picture is totally inverted. Results are clearly in favor of PLINGELING. It solves 235 instances (125 SAT and 110 UNSAT) whereas GLUCOSE-SYRUP *is only able* (GLUCOSE the underlying solver can only solve 173 instances (93 SAT and 80 UNSAT)) to solve 204 (105 SAT and 99 UNSAT). This is a big difference. However, if we study this result more deeply, we can observe that, here again, in many cases GLUCOSE-SYRUP is faster than PLINGELING. The differences between the two approaches arise after 500 seconds: PLINGELING is able to solve difficult and/or particular problems, by exploiting *inprocessing* techniques (xor-reasoning, equivalence checking, ...). The SAT'13 competition is indeed the first competition to contain so many problems with xor chains and counters. On this set of problem, *inprocessings* techniques are mandatory. However, for genuine parallel CDCL solvers, we showed that GLUCOSE-SYRUP is clearly a new parallel approach offering very good performances.

5 Conclusion

Clauses sharing among threads in a parallel SAT solver remains a difficult task. By experimentally studying learnt clauses usefulness, we propose a lazy policy for clauses exports. Furthermore, we propose a new scheme for clauses importations by putting them in probation before adding them to the clauses database. Experiments based on our solver GLUCOSE shows that this method for parallelization allows a very good scaling up of the underlying sequential engines.

References

1. Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, pages 200–213, may 2012.

2. Gilles Audemard, Benoît Hoessen, Said Jabbour, and Cédric Piette. An effective distributed D&C approach for the satisfiability problem. In *22nd Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'14)*, february 2014.
3. Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *proceedings of SAT*, pages 147–160, 2011.
4. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.
5. A. Biere, A. Cimatti, E. M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS'99*, pages 193–207, 1999.
6. Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 51, 2013. http://fmv.jku.at/lingeling.
7. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded asp solving with clasp. *TPLP*, 12(4-5):525–545, 2012.
8. Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel sat solving. In *proceedings of CP*, pages 252–265, 2010.
9. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *proceedings of IJCAI*, pages 499–504, 2009.
10. Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.
11. Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34(2):99–106, 2013.
12. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC*, pages 50–65, 2011.
13. Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer-Verlag.
14. Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *Automated Reasoning - 6th International Joint Conference*, pages 355–370, 2012.
15. George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. *AAAI'13*, 2013.
16. Stephan Kottler and Michael Kaufmann. SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.
17. Inês Lynce and Joao Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 136–141. Springer, 2006.
18. Matthew Moskewicz, Connor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.
19. Olivier Roussel. ppfolio. http://www.cril.univ-artois.fr/ roussel/ppfolio.