Revisiting Clause Exchange in Parallel SAT Solving

Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette

Revisiting Clause Exchange in Parallel SAT Solving

Gilles Audemard[1], Benoît Hoessen[1], Saïd Jabbour[1],
Jean-Marie Lagniez[2], and Cédric Piette[1] *

[1] Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,hoessen,jabbour,piette}@cril.fr
[2] Institute for Formal Models and Verification Johannes Kepler
University, AT-4040 Linz, Austria
jean-marie.lagniez@jku.at

Abstract. Managing learnt clause database is known to be a tricky task in SAT solvers. In the portfolio framework, the collaboration between threads through learnt clause exchange makes this problem even more difficult to tackle. Several techniques have been proposed in the last few years, but practical results are still in favor of very limited collaboration, or even no collaboration at all. This is mainly due to the difficulty that each thread has to manage a large amount of learnt clauses generated by the other workers. In this paper, we propose new efficient techniques for clause exchanges within a parallel SAT solver. In contrast to most of the current clause exchange methods, our approach relies on both export and import policies, and makes use of recent techniques that proves very effective in the sequential case. Extensive experimentations show the practical interest of the proposed ideas.

1 Introduction

The practical resolution of the SAT problem has received major attention these two last decades. Particularly, due to the wide availability of cheap multicore architectures, the focus is now on the development of efficient parallel engines, able to solve large real-world problems. Several of them have been recently proposed, e.g. `ManySat` [10], `SArTagnan` [12], `Plingeling` [4], `ppfolio` [16], `part-tree-learn` [11].

Portfolio schema is a possible approach to tackle parallelism. One idea of portfolio algorithms is the collaboration between the different workers. In the SAT case, their joint effort is mainly achieved through the exchange of learnt clauses. Unfortunately, it is very hard to predict whether a clause generated by a working thread will be useful for the others, or not. To deal with this problem, `ManySAT` proposes a dynamic clause sharing policy which uses pairwise size limits to control the exchange between threads [9]. However, most of implementations (`Plingeling` [4], `SArTagnan` [12], etc.) only share unit clauses with the other threads. Moreover, in the last SAT competition, a new portfolio called `ppfolio` obtained very good results; `ppfolio` is actually a simple script that runs different state-of-art sequential solvers in an independent way.

* This work has been supported by CNRS and OSEO, under the ISI project "Pajero".

Accordingly, no collaboration is achieved within this solver, and yet it proves very efficient during this competition. This shows that the current clause exchange techniques are not mature and may be improved.

The problem of predicting the usefulness of a given learnt clause is also known in the sequential case; recently, a new measure called *psm* [1] has been proposed to dynamically manage learnt clauses. Roughly, it consists in comparing the current (partial) interpretation to the set of literals of each learnt clause. The main idea is the following: if the set-theoretical intersection of the current interpretation and the clause is large, then the clause is unlikely useful in the current part of the search space. On the contrary, if this intersection is small, then the clause has a lot of chance to be useful for unit-propagation, reducing the search space. This measure has been used in a new strategy to manage the learnt clauses database which enables to *freeze* a clause, namely to remove it from the set of learnt clauses on a temporary basis, when it is considered "useless". Periodically all clauses are reevaluated in other to be frozen or activated. This technique proves very efficient in an empirical point of view, and succeeds to select relevant learnt clauses.

In this paper, we extend the results of [1] to the portfolio framework, proposing different efficient heuristical policies for exporting, importing and selecting relevant clauses for the different threads of a portfolio. This paper is structured as follows: in the next Section, we present the background knowledge about parallel SAT solving and learnt clause management. In the Section 3, we present some preliminaries about the behavior of a portfolio solver. Next, in Section 4, our case study solver called `PeneLoPe` is presented and it is compared to the best parallel SAT solvers in Section 5. Finally, we conclude with some perspectives.

2 Technical Background

First of all, we assume that the reader is familiar with Satisfiability notions (variables, literal, clause, unit clause, interpretations, `CNF` formula). Note that clauses and interpretations will be equally interpreted as set of literals. We just want to recall the global schema of CDCL (Conflict-Driven, Clause Learning) solvers: a typical branch of a CDCL solver can be seen as a sequence of decisions followed by propagations, repeated until a conflict is reached. Each decision literal, chosen by some heuristic, usually activity-based ones, is assigned at its own level, shared with all propagated literals assigned at the same level. Each time a conflict is reached, a *nogood* is extracted using a particular method, usually the First UIP (Unique Implication Point) one [14, 19]. The learnt clause is then added to the learnt clause database and a *backjumping* level is computed from it. The interested reader can refer to [7] for more details. In the rest of the section, we give background about important notions for the paper understanding.

Control of the learnt clauses database

The size of the learnt clauses database is clearly crucial to the solver performance. Indeed, keeping too many learnt clauses slows down the unit propagation process, while deleting too many of them breaks the overall learning benefit. To avoid such drawbacks,

solvers periodically remove some clauses considered to be useless. Consequently, identifying good learnt clauses - relevant to the proof derivation - is clearly an important challenge. The first proposed quality measure follows the success of the activity-based VSIDS heuristic. More precisely, a learnt clause is considered relevant to the proof, if it is often involved in recent conflicts *i.e.* frequently used to derive asserting clauses. Clearly, this deletion strategy supposes that a useful clause in the past could be useful in the future. In [2], another measure called *lbd* is used to estimate the quality of a learnt clause (we denote $lbd(c)$ the *lbd* value of a clause c). This new measure is based on the number of different decision levels appearing in a learnt clause and is computed when the clause is generated. Extensive experiments demonstrates that clauses with small *lbd* values are used more often than those with higher *lbd* ones. Note also that *lbd* of clauses can be recomputed when they are used for unit propagations, and updated if the it becomes smaller. This update process is important to get many good clauses. However, these both measures are obviously heuristical ones and solvers are not safe from regularly eliminating relevant learnt clauses.

Parallel SAT solving

Two approaches are commonly explored to parallelize SAT solvers. The first one is mainly a *divide-and-conquer* idea, which divides the search space into subspaces, successively allocated to SAT workers. Each time a worker finish its job (whereas the other ones are still doing their task), a load balancing strategy is invoked, and dynamically transfers subspaces to this idle worker [5, 6]. A closely related approach is the iterative partitioning one [11]. Note that some of these approaches are able to share clauses [17, 11] between workers. On the other hand, the parallel portfolio strategy exploits the complementarity between different sequential CDCL strategies to let them compete and cooperate on the same formula [10, 4, 12]. Since each worker deals with the whole formula, there is no need to introduce load balancing overheads, and cooperation is only achieved through the exchange of learnt clauses. With this approach, the crafting of the strategies is important, especially with a small number of workers. In general, the objective is to cover the space of good search strategies in the best possible way. Such strategies are efficient on multicore architectures.

As we said above, the size of the learnt clause database is crucial for sequential solvers. Then, for a parallel portfolio SAT solver, it is not desirable to share all learnt clauses between all threads. To deal with this problem, one has to select carefully the clauses that a thread wants to share with the others. A natural solution is to take into account the size of learnt clauses and share only the smallest ones (size less than 8 for example [10]). Based on the observation that small clauses appears less and less during the search, authors of `ManySAT` propose a dynamic clause sharing policy which uses pairwise size limits to control the exchange between threads [9].

However, it is surprising that `Plingeling`, one of the winner of the SAT'11 competition shares between threads only unit clauses. Furthermore, `ppfolio` which have obtained good results in that competition runs different state-of-art sequential solvers in an independent way without any sharing. This last observations show that current clause exchange techniques are not mature and may be improved. This is one of the goal of this paper.

3 Parallelism, Collaboration and Clause Exchange: a Premilinary Experimentation

To illustrate the current behavior of portfolio solvers with respect to clause exchanges, we first have conducted preliminaries experiments on a state-of-the-art solver. For a sequential solver, a "good" learnt clause is a clause that is used during the unit propagation process and the conflict analysis. For portfolio solvers, one can quite safely state the same idea: a "good" shared clause is a clause that helps at least one other thread reducing its search space, namely *propagating*.

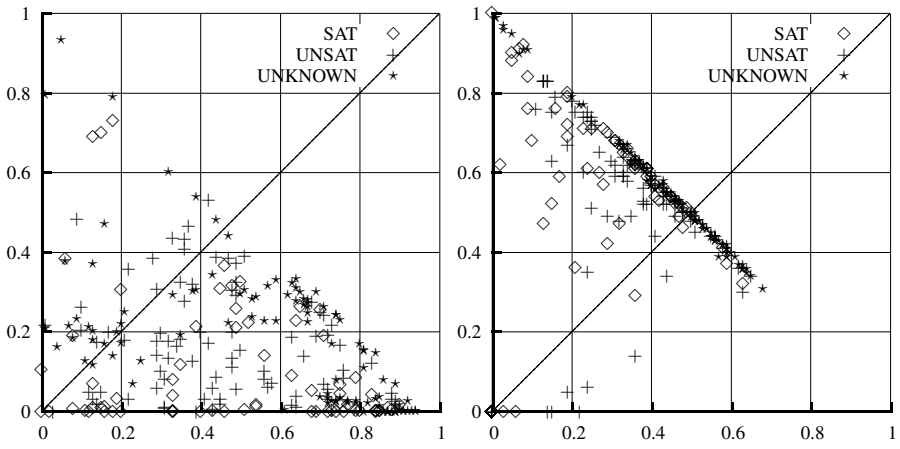
Accordingly, we wanted to know how useful are the clauses shared in a portfolio-based solver. To this end, we ran some experiments[3] using a state-of-the-art portfolio-based SAT solver. We choose the solver `ManySAT` 2.0 (based on `Minisat 2.2`), because in this solver, the only difference between the working threads are caused by the first decision variables which are selected randomly. Except this initial interpretation, each DPLL worker exhibits the exact same behavior (in terms of restart strategy, branching variable heuristics, etc.), which allows us to make a fair comparison about clause exchange without any side effect. Hence, it represents a good framework to deal with parallel SAT solvers and clauses sharing. By default all clauses of size less than 8 are shared. Moreover, `ManySAT` provides a deterministic mode [8]. Let us emphasize that we have activated this option to make the obtained results fully reproducible and we report the detailled results online[4].

Let $\mathcal{SC}$ be the set of shared clauses, namely the set-theoretical union of each clause exported by a given thread to all the other ones. In this experiment, for each thread, we have considered two particular kinds of shared clauses. First, the shared clauses that are actually used (at least once) by a working thread to propagate. We denote this set $used(\mathcal{SC})$. Second, we have also focused on the set of shared clauses that are deleted without having been from any help during the search. This set is denoted $unused(\mathcal{SC})$. Clearly, $\mathcal{SC} \backslash (used(\mathcal{SC}) \cup unused(\mathcal{SC}))$ represents the set of clauses that have neither been used nor been deleted, yet.

Figure 1 synthetizes the results obtained during this first experiment. The results are reported in the following way: each point of Figure 1 is associated with an instance, and the x-axys corresponds to the rate $\#used(\mathcal{SC})/\#\mathcal{SC}$, whereas the y-axys corresponds to the rate $\#unused(\mathcal{SC})/\#\mathcal{SC}$, and we report the average rate over the different threads. Figure 1(a) gives the results for `ManySAT`. First of all, we can remark that the rate of useful shared clauses differs greatly over different instances. We can also note, that in a lot of cases, `ManySAT` keeps shared clauses during the entire search (dots near the x-axys). This is due to the non-aggressive cleaning strategy of `Minisat` where in many instances no cleaning are performed. Threads can keep useless clauses a long time and have to support an over cost without any benefit.

[3] All experimentations of this paper have been conducted on a dual socket `Intel XEON X5550` quad-core 2.66 GHz with 8 MB of cache and a RAM limit of 32GB, under Linux CentOS 6 (kernel 2.6.32). All solvers use 8 threads. The timeout was set to 1200 seconds WC for each instance. If no answer was delivered within this amount of time, the instance was considered unsolved. We used the application instances (300) of the SAT competition 2011.

[4] `http://www.cril.fr/~hoessen/penelope.html`



(a) `ManySAT` (b) aggressive `ManySAT`

Fig. 1. Comparison between useful share clauses and unused deleted clauses. Each dot corresponds to an instances. x-axis gives the rate of useful shared clauses $\#used(\mathcal{SC})/\#\mathcal{SC}$, whereas the y-axis gives the rate of unused deleted shared clauses $\#unused(\mathcal{SC})/\#\mathcal{SC}$.

We have conducted the same experiment with a much more agressive cleaning strategy. We have choosed the one presented in [1] (see Section 4) and we report the result in Figure 1(b). Here, in many cases, shared clauses are deleted without any usage and the percentage of shared clauses that are used at least one time decreases with respect to the basic version of `ManySAT`. These results can be explained quite easily. If only few cleanings are done, the threads have to manage a lot of useful *and* useless shared clauses. In one hand, it owns a lot of information about the problem to solve, and propagates many units clauses. On the other hand, such a solver has to maintain a large number of clauses uselessly, which greatly slows down its exploration.

Conversely, when many cleanings are achieved, another problem occurs. Indeed, if a given clause is not used in conflict analysis and/or unit propagation very often, it has then a lot of chances to be quickly removed. Therefore, threads using an agressive strategy spend a lot of time importing clauses that are never used. We can also notice that only using the *lbd* measure for clause usefulness seems not efficient. Indeed, shared clauses are here small clauses, so they have small *lbd* values. Even if we can try to tune the cleaning strategy to obtain a stronger solver, we think that the classical strategy used to manage learnt clauses is not appropriate in the case of clauses sharing and multicore architectures. We propose a new scheme in the next section.

4 Selecting, Sharing and Activating Good Clauses

Managing learnt clauses is known to be difficult in the sequential case. Furthermore, dealing with imported clauses from other threads leads to additional problems:

– Imported clauses can be subsumed by clauses already present in the database. Since subsumption computation is time consuming, it is necessary to give the possibility to remove periodically learnt clauses.
– Imported clauses may be useless during a long time, and suddenly become useful.
– Each thread has to manage many more clauses.
– Characterizing good imported clauses is a real challenge.

For all of these reasons, we propose to use the dynamic management policy of learnt clauses proposed in [1] inside each thread. This recent technique enables to activate or freeze some learnt clauses, imported or locally generated. The advantage is twofold. The overhead caused by imported clauses is greatly reduced since clauses can be frozen. Nevertheless, clauses estimated useful in the next future of the search are activated. Let us present more precisely this method in the next Section.

Freezing clauses

The strategy proposed in [1] differs from the other ones proposed in the past (see Section 2). Indeed, it is based on a dynamic freezing and activation principle of learnt clauses. At a given search state, it activates the most promising learnt clauses while freezing irrelevant ones. In this way, learned clauses can be discarded from the current step, but may be activated again in future steps of the search process. This strategy cannot be used with the other known measures such as activity or *lbd*-based ones. Indeed, the activity (VSIDS-based) measure is dynamic but can only be used to update the activity of learnt clauses currently in the database, while the *lbd* value of a given learnt clause is either static (and does not change during search) or dynamic but, in this case, the same problem as VSIDS-based one occurs. Then, this strategy is associated with another measure, defined in the following, for identifying good learnt clauses [1].

Let Σ be a CNF formula, c be a clause learnt by the solver, and ω the current interpretation saved from the polarity choice of decision variables [15]. The *psm* value of the clause c w.r.t. ω, denoted $psm_\omega(c)$, is equal to:

$$psm_\omega(c) = |\omega \cap c|$$

psm is a highly dynamic measure, since it is mainly based on the current interpretation. It aims at selecting relevant context (i.e. learnt clauses) with respect to the search in progress. To this end, the clauses that exhibit a low *psm* are considered relevant. Indeed, the lower is a *psm* value, the more likely the related clause is about to unit-propagate some literal, or to be falsified. On the opposite, a clause with a large *psm* value has a lot of chance to be satisfied by many literals, making it irrelevant for the search in progress.

Thus, only clauses that exhibit a low *psm* are selected and currently used by the solver, the other clauses being *frozen*. When a clause is frozen, it is removed from the list of the watched literals of the solver, in order to avoid the computational overhead of maintaining the data structure of the solver for this useless clause. Nevertheless, a frozen clause is not erased but it is kept in memory, since this clause may be useful in the next future of the search. As the current interpretation evolves, the set of learnt clauses actually used by the solver evolves, too. In this respect, the *psm* value is computed

periodically, and sets of clauses are frozen or unfrozen with respect to their freshly computed new value.

Let P_k be a sequence where $P_0 = 500$ and $P_{i+1} = P_i + 500 + 100 \times i$. A function "*updateDB* " is called each time the number of conflict reaches P_i conflicts (where $i \in [0..\infty]$). This function computes new *psm* values for every learnt clauses (frozen or activated). A clause that has a *psm* value less than a given limit l is activated in the next part of the search. If its *psm* does not hold this condition, then it is frozen. Moreover, a clause that is not activated after k (equal to 7 by default) time steps is deleted. Similarly, a clause remaining active more than k steps without participating to the search is also permanently deleted.

Given the *psm* and *lbd* measures, we now define different policies for clause exchange. In a typical CDCL procedure, a nogood clause is learnt after each conflict. It appears that all clauses cannot be shared, especially because some of them are not useful in a long term. So, when collaboration is achieved, this is limited through some criterion. To the best of our knowledge, in all current portfolio solvers, this criterion is only based on the information from the sender of the clause, the receiver having to accept any clause judged locally relevant by another worker.

We present in the next Section a technique where both the sender and the receiver of a clause havea strategy. Obviously, any sender (export strategy) tries to find in its own learnt clause database the most relevant information to help the other workers. However, the receiver (import strategy) here does not accept the shared clauses in a blind way. We have called our case study solver `PeneLoPe`[5] (**P**arallel **L**bd **P**sm solver).

Importing clause policy When a clause is imported, we can consider different cases, depending on the moment the clause is attached for participating to the search.

- *no-freeze*: each imported clause is actually stored with the current learnt database of the thread, and will be evaluated (and possibly frozen) during the next call to *updateDB* .
- *freeze-all*: each imported clause is *frozen* by default, and is only used later by the solver if it is evaluated relevant w.r.t. unfreezing conditions.
- *freeze*: each imported clause is evaluated as it would have been if locally generated. If the clause is considered relevant, it is added to the learnt clauses, otherwise it is frozen.

Exporting clause policy Since `PeneLoPe` can freeze clauses, each thread can import more clauses than it would with a classical management of clauses, where all of them are attached. Then, we propose different strategies, more or less restrictive, to select which clauses have to be shared:

- *unlimited*: any generated clause is exported towards the different threads.
- *size limit*: only clauses whose size is less than a given value (8 in our experiments) are exported [9].

[5] in reference to Odysseus's faithful wife who wove a burial shroud, linking many *threads* together

psm used	export strategy	restart strategy	import strategy	#SAT	#UNSAT	#SAT + #UNSAT
✓	*lbd limit*	*lbd*	*no freeze*	94	111	**205**
✓	*lbd limit*	*lbd*	*freeze*	89	**113**	202
✓	*size limit*	*lbd*	*freeze*	93	107	200
✓	*size limit*	*lbd*	*no freeze*	89	107	196
✓	*size limit*	*luby*	*no freeze*	**97**	98	195
✓	*lbd limit*	*lbd*	*freeze all*	89	102	191
✓	*size limit*	*luby*	*freeze all*	96	92	188
✓	*unlimited*	*lbd*	*freeze*	86	102	188
✓	*size limit*	*luby*	*freeze*	92	96	188
✓	*lbd limit*	*luby*	*freeze*	91	97	188
ManySAT	-	-	-	95	93	188
✓	*lbd limit*	*luby*	*no freeze*	90	94	184
✓	*unlimited*	*luby*	*freeze*	91	92	183
	size limit	*luby*	*no freeze*	92	90	182
✓	*unlimited*	*luby*	*no freeze*	89	88	177
✓	*size = 1*	*lbd*	*freeze*	89	88	177

Table 1. Comparison between import, export & restart strategies using deterministic mode

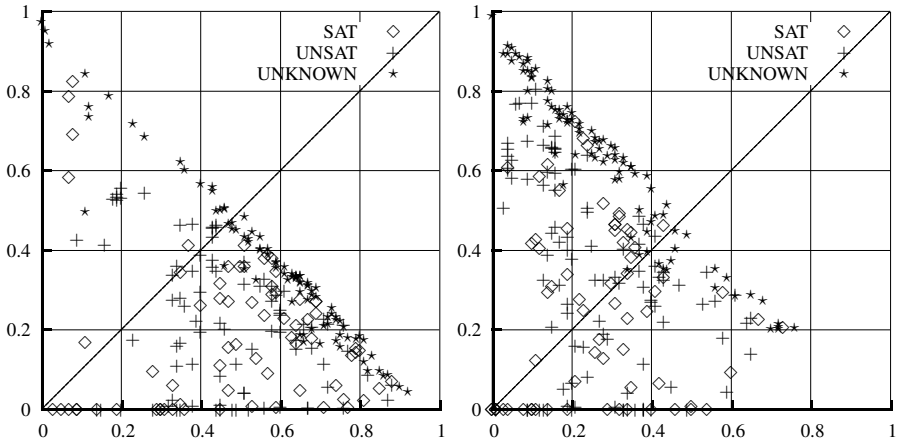
– *lbd limit*: a given clause c is exported to other threads if its *lbd* value $lbd(c)$ is less than a given limit value d (8 by default). Let us also note that the *lbd* value can vary over time, since it is computed with respect to the current interpretation. Therefore, as soon as $lbd(c)$ is less than d, the clause is exported.

Restarts policy Beside exchange policies, we define two restart strategies.

– *Luby*: Let l_i be the i^{th} term of the Luby serie[13]. The i^{th} restart is achieved after $l_i \times \alpha$ conflicts (α is set to 100 by default).
– *LBD* [2]: Let LBD_g be the average value of the LBD of each learnt clause since the beginning. Let LBD_{100} be the same value computed only for the last 100 generated learnt clause. With this policy, a restart is achieved as soon as $LBD_{100} \times \alpha > LBD_g$ (α is set to 0.7 by default). In addition, the VSIDS score of variables that are unit-propagated thank for a learnt clause whose *lbd* is equal to 2 are increased, as detailled in [2].

We have conducted experiments to compare these different import/export/restart strategies. We ran these different versions and Table 1 presents a sample of the obtained results This table report for each strategy the number of SAT instances solved (#SAT), together with the number of UNSAT instances solved (#UNSAT) and total (#SAT + #UNSAT).

Let us take a first look at the export strategy. Unsurprisingly, the "unlimited" policy obtained the worst results. Indeed, none of these versions have been able to solve more than 190 instances, regardless all other policies (export, restart). Here, every generated clause is exported, and we reach the maximum level of communication. As expected, with the multiplicity of the workers, the solvers are soon overwhelmed by clauses and their performances drop.



(a) *size limit* + *luby* + *no freeze* (SLN) (b) *lbd limit* + *lbd* + *freeze* (LLF)

Fig. 2. Comparison between usefull share clauses and useless deleted clauses. Each dot corresponds to an instances. x-axis gives the rate of useful shared clauses $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, whereas the y-axis gives the rate of unused deleted shared clauses $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

This was the reason why a size-based limit was introduced with the idea that the smallest clauses produce the best syntactic filtering, and therefore are preferable. Indeed, in Table 1, it appears clearly that "size limit" (clauses containing less than 8 literals) policy outperforms the "unlimited" one. This simple limit shows its usefulness, but a main drawback is that it has been shown [3] that longer clauses may greatly reduce the size of the proof.

Using the *lbd* value $lbd(c)$ of a clause c can improve the situation as $lbd(c) \leq \text{size}(c)$. Hence, if the same value v is used for both the size and the *lbd* limits, more clauses are exported with the *lbd* policy. So, specifying a limit on the *lbd* allows us to import larger clauses if those ones are heuristically considered as promising. This could represent a problem for a parallel solver without the ability to freeze some clauses. Nevertheless, as PeneLoPe contains such mechanism, the impact is greatly reduced. From an empirical point of view, Table 1 shows that the "lbd limit" obtains the best results among all exporting strategies. We have also tried to limit the export to unary clauses (line *size=1*) like most current portfolio solvers do, but this does not lead to good performance, since only 177 instances are solved.

Let us now focus on the restart strategy. Quite obviously, the "luby" technique obtains overall worst results than the "*lbd* " one. This clearly shows the particular interest of this *lbd* concept introduced in [2]. About import strategy, no clear winner appears when looking at the results in Table 1. Indeed, the best results in term of number of solved SAT instances is obtained with *no freeze* (97) when associated with the "luby" restart and the "size limit" export strategy, whereas the best number of solved UNSAT instances is obtained with the *"freeze"* strategy (113). Furthermore, *no freeze* enables to obtain the best overall result solving 205 instances out of the 300 used ones. Hence,

it would be audacious to plead for one of the 3 proposed techniques. However, a large number of our proposed policies performs in practice better than "classical" clause exchange techniques, represented in Table1 by `ManySat`.

In a second experiment, we wanted to assess the behavior of the solver when using some of our proposed policies. To this end, we have conducted the exact same experiments than the one presented in Section 3; the obtained results are reported in the Figure 2. First, we have tried with *size limit*, *luby*, and *no freeze* policies (denoted `SLN`, see Figure 2(a)). Clearly, this version behaves very well, since most of the dots are located under the diagonal. Moreover, for most instances, $\frac{\#usedSC+\#unused(SC)}{\#SC}$ is close to 1 (dots near the second diagonal), which indicates that the solver does not carry useless clauses without deleting them. Most of them proves useful, and the other ones are deleted.

Then, the experimentation was conducted with the *lbd* limit, *lbd* and *freeze* combo (denoted `LLF`, see Figure 2(b)). At first sight, the behavior is here less satisfying than the `SLN` version, since for most instances at least half of imported clauses are deleted without being from any help. Actually, in this version, a much larger number of clauses are exported due to the "*lbd* limit" export policy, which leads to a lower rate of useful clauses. Fine-tuning parameters (*lbd* limit values, number of time a clause has to be frozen before being permanently deleted, etc.) might improve this behavior.

Looking at some detailled statistics provided in Table 2, it indeed appears that the `LLF` version shares a lot more clauses than the `SLN` one (column nb_u). Note that this Table contains some other very interesting information. For instance, it enables to see that for some benchmarks (e.g. `AProVE07-21`), about 90% of imported clauses are actually frozen and do not immediately participate to the search, whereas for other instances, we face the opposite situation (`hwmcc10...`) with only 10% of clauses that are frozen when imported. This reveals the high adaptability of the *psm* measure. Let us focus now on the number of imported clauses, compared to the number of conflicts needed to solve the instance. The `SLN` version produces very often more conflict clauses than it imports from other working threads ($nb_c/nb_i < 1$), even though this is not true with some benchmarks (e.g. `AProVE07-21`, `hwmcc10...`). Note that the nb_c/nb_i rate of the `LLF` version exhibits a very high variability, from 0.58 for the smallest value in Table 2 (`velev-pipe-o-uns...`) to more than 4, meaning that in such cases, each time the solver produces a conflict (and consequently a clause), it imports more than 4 clauses on average. Let us also emphasize that the computationnal cost of the *psm* measure is not major (see "*psm* time" column). During all our experiments, `PeneLoPe` have spent at most 5% of the solving time to compute *psm*.

On a more general view, even if the *no-freeze* policy seems to be the best in terms of efficiency in communication between threads of the solver, it has the disadvantage of adding every imported clause in the set of active clauses. This leads to a lower number of propagation per second until the next re-examination of the whole clause database. This might be a problem if we want to increase the number of threads of the solver. On the other hand, the *freeze-all* policy does not slow down the solver. Yet, such solver is not able to use the imported clauses as soon as they are available, and therefore explores subspaces that would have been pruned with the *no-freeze* policy.

instance	version	time	nb_c	$nb_i (nb_c/nb_i)$	nb_f	nb_u	nb_d	*psm* time
dated-10-17-u	SLN	TO	1771	278 (0.15)	0%	45%	49%	2%
	LLF	949	1047	1251 (0.83)	64%	20%	60%	4%
hwmcc10-...	SLN	TO	5955	7989 (1.34)	0%	35%	60%	3%
k50-eijkbs6669-tseitin	LLF	766	3360	15299 (4.55)	10%	11%	80%	5%
velev-pipe-o-uns-1.1-6	SLN	150	981	69 (0.07)	0%	60%	24%	2%
	LLF	48	296	173 (0.58)	41%	31%	33%	3%
sokoban-sequential-p145-	SLN	TO	182	86 (0.47)	0%	92%	4%	0.1%
microban-sequential.040	LLF	530	74	155 (2.09)	5%	58%	17%	0.4%
AProVE07-21	SLN	10	78	83 (1.06)	0%	35%	16%	3%
	LLF	31	143	506 (3.53)	89%	9%	57%	5%
slp-synthesis-aes-bottom13	SLN	445	1628	194 (0.11)	0%	58%	30%	3%
	LLF	91	309	298 (0.96)	71%	24%	49%	4%
velev-vliw-uns-4.0-9-i1	SLN	TO	1664	262 (0.15)	0%	55%	40%	2%
	LLF	906	1165	824 (0.70)	35%	37%	48%	5%
x1mul.miter...-359	SLN	819	2073	421 (0.20)	0%	51%	37%	5%
	LLF	280	680	1134 (1.66)	76%	16%	59%	5%

Table 2. Statistics about some unsatisfiable instance solving. For each instance and each version, we report the WC time needed to solve it, the number of conficts (nb_c, in thousands), the number of imported clauses (nb_i, in thousands) with between brackets the rate between nb_i and nb_c, the percentage of clauses frozen at the import (nb_f), the percentage of useful imported clauses (nb_u) and the percentage of unused deleted clauses (nb_d). Finally, we provide the rate of time (w.r.t. the overall solving time) spent on computing the *psm* value. Except for time, we compute the average between the 8 threads for these statistics.

5 Comparison with state of the art solvers

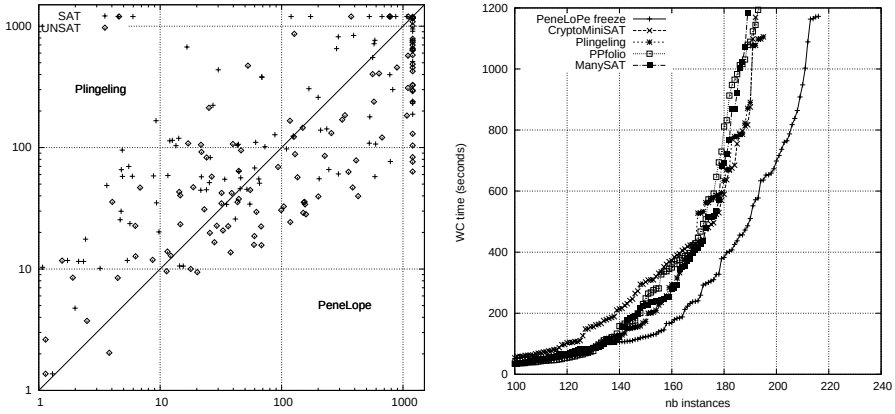
In this Section, we propose a comparison of two of our proposed prototypes against state-of-the-art parallel SAT solvers. We have selected solvers that prove the most effective during the last competitive events: `ppfolio` [16], `cryptominisat` [18], `plingeling` [4] and `ManySat` [10].

For `PeneLoPe`, we choose for both versions the *lbd* restart strategy and the *lbd limit* for the export policy. These two versions only differ from their import policies: *freeze* and *no freeze*. Let us precise that contrary to previous experiments, we do not use the deterministic mode in these experiments, in order to obtain the best possible performance.

Figure 3 shows the obtained results through different representations; Table 3(a) provides the number of solved instances for the different solvers, Figure 3(b) details the comparison of `PeneLoPe` and `Plingeling` through a scatter plot, and a cactus plot in Figure 3(c) gives the number of solved instances w.r.t. the time (in seconds) needed to solve them. `PeneLoPe` outperforms all other parallel solvers; indeed, it succeeds to solve 216 instances while no other solver is able to exceed 200 (Table 3(a)). Note that only considering SAT instances, the best results come from `plingeling` which solves 99 instances. This is particularly noticeable in Figure 3(b) where `PeneLoPe` and `plingeling` are more precisely compared; indeed, most of "SAT dots" are located above the diagonal, illustrating the strength of `plingeling` on these instances. How-

Solver	#SAT	#UNSAT	#SAT+#UNSAT
PeneLoPe *freeze*	97	**119**	**216**
PeneLoPe *no freeze*	96	**119**	215
plingeling [4]	**99**	97	196
ppfolio [16]	91	103	194
cryptominisat [18]	89	104	193
ManySat [10]	95	92	187

(a) PeneLoPe VS state-of-the-art parallel solvers



(b) PeneLoPe *freeze* VS Plingeling (c) Cactus plot

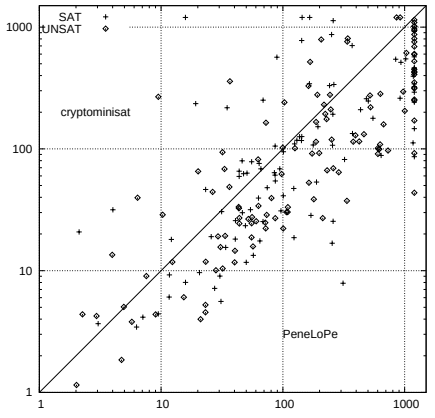
Fig. 3. Comparison on 8 cores

ever, results for SAT instances are closer from each other (97 for PeneLoPe *freeze*, 95 for ManySat, etc.), the gap being more important for UNSAT problems.

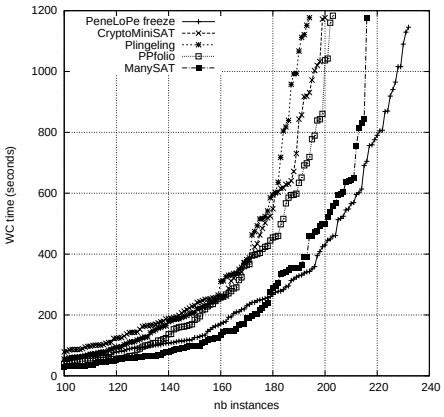
In addition, we have compared the same solvers on a 32 cores architecture. More precisely, the considered hardware configuration is now Intel Xeon CPU X7550 (4 processors, 32 cores) 2.00GHz with 18 MB of cache and a RAM limit of 256GB. The software framework is the same as with previous experiments. Each solver is run using 32 threads, and the obtained results are displayed in Figure 4 in a similar way than previously. First, let us remark that except for plingeling, all solvers improve their results when they are run with a larger number of threads. The benefit is limited for certain solvers, however. For example, cryptominisat solves 193 instances with 8 threads, and 201 instances with 32 threads. The improvement is stronger with PeneLoPe whose both versions solve 15 extra instances when 32 threads are used, and especially for ManySAT with a gain of 29 instances. The gap can be more remarkable looking at the cactus plot in Figure 4(c), since our 3 competitors solve about the same number of instances within the same time (curves very close to each other), whereas the curve of PeneLoPe and ManySAT clearly shows their ability to solve a larger number of instances within a more restricted time. Besides, it is worth noting that PeneLoPe solves the same number of instances as Plingeling, ppfolio and cryptominisat with a (virtual) time limit of only 400 seconds. Finally, we can also notice than PeneLoPe can be improved on SAT instances. Indeed, it appears that

Solver	#SAT	#UNSAT	#SAT+#UNSAT
PeneLoPe *freeze*	104	127	**231**
PeneLoPe *no freeze*	99	**131**	230
ManySAT [10]	105	111	216
ppfolio [16]	**107**	97	204
cryptominisat [18]	96	105	201
Plingeling [4]	100	95	195

(a) PeneLoPe VS state-of-the-art parallel solvers



(b) PeneLoPe *freeze* VS cryptominisat



(c) Cactus plot

Fig. 4. Comparison on 32 cores

Luby restarts are more efficient for SAT than for UNSAT, whereas the exact opposite phenomenon happens for UNSAT instances with the *lbd* restart strategy.

Adding computing units has different impacts. For instance, for ppfolio and plingeling the gain is not major, since augmenting the number of working threads "just" improves the number of CDCL sequential solvers that explore the search space; each worker does not benefit from the exploration of the other ones, since with these solvers, little (if any) collaboration is done. PeneLoPe benefits more from more computing units because the number of exchanged clauses coming from different search subspaces is greater. This leads to a wider knowledge for each thread without being slowed down too much, thanks to the freezing mechanism.

Finally, let us emphasize that during all our experiments with PeneLoPe, all working threads share the exact same parameters and strategies, just like in our preliminary experimentation in Section 3. Improving diversification in the different sequential CDCL searches should probably boost even more our case study solver.

6 Conclusion

In this paper, we have proposed new strategies to manage clause exchange within parallel SAT solvers. Based on the recent *psm* and *lbd* concepts, the idea is to adopt different strategies for import and export of clauses. We have carrefully studied different empir-

ical aspects of our proposed ideas and compared our solver to best known parallel SAT engines, showing that it appears to be a highly competitive prototype.

Clearly, diversifying the different working threads should improve the performance of our case study solver `PeneLoPe`, since this technique is known to be the cornerstone of the efficiency of some portfolio solvers, like `ppfolio`. We plan to study this point in the next future.

References

1. Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *proceedings of SAT*, pages 147–160, 2011.
2. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.
3. Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
4. Armin Biere. (p)lingeling. `http://fmv.jku.at/lingeling`.
5. Wahid Chrabakh and Rich Wolski. GrADSAT: A parallel SAT solver for the grid. Technical report, UCSB, 2003.
6. Geoffrey Chu, Peter J. Stuckey, and AAron Harwood. Pminisat: a parallelization of minisat 2.0. Technical report, SAT Race, 2008.
7. Adnan Darwiche and Knot Pipatsrisawat. *Complete Algorithms*, chapter 3, pages 99–130. IOS Press, 2009.
8. Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Saïs. Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.
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. Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *proceedings of CP*, pages 385–399, 2011.
12. Stephan Kottler and Michael Kaufmann. SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.
13. Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. In *proceedings of ISTCS*, pages 128–133, 1993.
14. Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *proceedings of ICCAD*, pages 220–227, 1996.
15. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.
16. Olivier Roussel. ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio`.
17. Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.
18. Mate Soos. Cryptominisat. `http://www.msoos.org/cryptominisat2/`.
19. Lintao Zhang, Connor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285, 2001.