Master Thesis

Sharing Information in Parallel Search with Search Space Partitioning

Davide Lanti

03. April 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Künstliche Intelligenz
Professur für Knowledge Representation and Reasoning

Betreut von:
Prof. Dr. rer. nat. Steffen Hölldobler
Dipl.-Inf. Norbert Manthey

Davide Lanti
Sharing Information in Parallel Search with Search Space Partitioning
Master Thesis, Fakultät für Informatik
Technische Universität Dresden, April 2013

Task of the Master Thesis

Surname, Name:	Lanti, Davide
Course of Studies:	EMCL
Matrikelnummer:	3725496
Title:	***Sharing Information in Parallel Search with Search Space Partitioning***
Task Description:	The question whether a propositional formula in conjunctive normal form is satisfiable (SAT) is answered with powerful clause learning SAT solvers. Parallel solvers can be categorized into competitive and cooperative systems. Most of the recent solvers belong to the first group and there has been much research on improving these solvers. A famous class of solvers in this category are portfolio solvers, which have been improved with complex clause exchange methods. However, solvers of the latter category are believed to scale better with the number of available cores. In this thesis, existing clause sharing mechanism for cooperative search space partitioning SAT solvers should be examined and improved. Finally, the clause sharing mechanisms should be compared against each other.

Betreuer:		Davide Lanti
verantwortlicher Hochschullehrer:		Prof. Dr. rer. nat. Steffen Hölldobler
Institut:		Künstliche Intelligenz
Lehrstuhl:		Knowledge Representation and Reasoning
Beginn am:		01.10.2012
Einzureichen am:		03.04.2013

Abstract. Recent computing architecture turned parallel. A single CPU now provides up to 16 cores. These computing resources should also be exploited for solving search problems, e.g. the well researched SAT problem. In this thesis a new technique for sharing information among several computing units of a search space partitioning parallel SAT solver is presented. The approach is correct, in the sense that each solver working in parallel can receive clauses only if these are a semantic consequence of the problem it is assigned to solve. The approach is valuable since it outperforms currently available techniques for clause sharing both in terms of number of shared clauses and performance. Evaluation has been performed on two different benchmarks, with two different versions of the solver. On the first benchmark, consisting of 600 instances, the new approach permits to solve 12 more instances out of 600 than both non-sharing and existing state-of-the-art sharing techniques. The second benchmark, consisting of 292 instances, has been submitted to an improved version of the solver that takes into account memory consumption issues made evident during the tests on the first benchmark. As a result, 17 more instances are solved when compared against non-sharing, and at least 9 more instances are solved when compared against existing clause sharing techniques.

Contents

1 Introduction **9**

2 Basics **12**

3 Sequential SAT Solving **15**
- 3.1 Abstract Reduction Systems . . . 15
- 3.2 $\mathcal{CDCL}$ Abstract Reduction System . . . 16
- 3.3 Modern CDCL Solvers . . . 18
  - 3.3.1 Clause Learning . . . 18
  - 3.3.2 VSIDS Decision Heuristic . . . 22
  - 3.3.3 Two-watched Literals Unit Propagation . . . 23
  - 3.3.4 Learnt Clauses Database Management . . . 23

4 Parallel SAT Solving **25**
- 4.1 Evaluation of Parallel Procedures . . . 25
- 4.2 Related Work on Parallel SAT Solving . . . 26
- 4.3 Search Space Partitioning Approaches . . . 28
- 4.4 Iterative Partitioning . . . 29
- 4.5 Clause Sharing so Far . . . 31
- 4.6 Clause Sharing in Iterative Partitioning Solvers . . . 34
  - 4.6.1 Assumption-based Clause Tagging . . . 35
  - 4.6.2 Flag-based Clause Tagging . . . 36
- 4.7 Position-based Clause Tagging: a New Approach to Clause Sharing . . . 38

5 Empirical Evaluation **41**
- 5.1 Underlying Iterative Partitioning Solver . . . 41
  - 5.1.1 VSIDS Scattering . . . 41
- 5.2 Clause Sharing Implementation . . . 42
  - 5.2.1 From Position-Based Clause Tagging to Level-Based Clause Tagging . . . 42
  - 5.2.2 Pools of Shared Clauses . . . 42
  - 5.2.3 Handling of Unit Clauses . . . 43
  - 5.2.4 Pulling Clauses . . . 44
  - 5.2.5 Pushing Clauses . . . 44
- 5.3 Instances Selection . . . 45
- 5.4 Tests on SAT12 . . . 45
  - 5.4.1 Quantitative Analysis on Solved and Unsolved Instances . . . 47
  - 5.4.2 Clause Sharing Analysis . . . 49
  - 5.4.3 Partition Tree and Resource Consumption Analysis . . . 50
  - 5.4.4 Scalability Analysis . . . 52
- 5.5 Tests on SAT09 . . . 53
  - 5.5.1 Quantitative Analysis on Solved and Unsolved Instances . . . 55
  - 5.5.2 Clause Sharing Analysis . . . 56

5.5.3 Partition Tree and Resource Consumption Analysis 57
5.5.4 Efficiency and Speedup Analysis 58

6 Conclusion and Future Work **59**

References **61**

1 Introduction

With every probability, the most interesting class of problems in theoretical computer science is the class of Non-deterministic polynomial problems ($\mathcal{NP}$), introduced for the first time by Cook in [Coo71] during the proceedings of the ACM Symposium on Theory of Computing. That is an important event since the most famous, and still open, question in theoretical computer science was posed for the first time: are $\mathcal{P}$[1] and $\mathcal{NP}$ the same object?

Being the first problem recognised as a member of $\mathcal{NP}$ class, SAT problem is the prince problem of theoretical computer science and a lot of literature has been published on it. Yet, for long time this interest has been more circumscribed to the goal of proving $\mathcal{NP}$-completeness of other problems, rather than tailored towards the goal of SAT solving. One has to keep in mind, however, that research on SAT solving was long active before SAT was recognised as a $\mathcal{NP}$ problem, as the first version of the currently state-of-the-art structured SAT solver (DPLL) was introduced in sixties [DLL62].

Situation dramatically changed in the past few years. Enhancements to DPLL procedure, most remarkably the clause learning enhancement [SS96], led to such an improvement of basic DPLL that nowadays CDCL solvers, that are DPLL solvers with clause learning feature, can timely solve instances with million of clauses and thousands of variables. Still, SAT is in $\mathcal{NP}$ and thus one cannot expect, unless $\mathcal{P}$ and $\mathcal{NP}$ are the same class, that CDCL solvers are always effective. Indeed they are not, for example they are extremely ineffective in solving random k-sat instances where the ratio between the number of clauses and the number of propositional variables is near the *phase transition* limit [AAA$^+$08].

Luckily enough, practical industrial problems are "easy" w.r.t. the aforementioned measure, and CDCL procedures are successfully used to solve a number of different search problems, like railway scheduling [GHM$^+$12], vehicle routing [Goe10], planning [KS92], software verification [DKW08], and many others. As far as the solution needs not to be optimal, every ($\mathcal{NP}$) constraint satisfaction problem can be encoded into a SAT problem, and then be solved with a SAT solver.

The first parallel DPLL-based SAT solver was presented in 1996 [BS96] in a work published on the Annals of Mathematical and Artificial Intelligence. Parallel computation facilities were provided by an underlying grid. At that time, indeed, grids were the only environment where a parallel procedure could be deployed, thus the majority of these first SAT solvers were using grids as backbone.

Things dramatically changed in the past few years (more precisely, since the "thermal wall" for CPUs was hit in 2004), and affordable parallel multi-core architectures are nowadays available and widespread. This pones a challenge to programmers, since parallel procedures are harder to design and consequently it is more difficult to exploit parallel resources effectively. With this respect, a parallel SAT solver makes no exception.

Attempts to parallelize DPLL procedure itself have been considered in [Kas90].

[1] Class of problems that can be decided in polynomial deterministic time

There it has been proved that *unit-propagation*, that is the most important deterministic variable decision technique used in modern SAT solvers, is $\mathcal{P}$-complete. In other words, unit propagation is "inherently sequential", and thus it cannot be parallelized effectively. This is somehow a bad news, since around 80% of total run-time of a modern CDCL solver is spent in unit-propagation steps [Man11]. Still, work in [Man11] shows that, for practical instances, even unit propagation can be parallelized with a certain degree of success. Unfortunately, the approach there described does not scale well beyond two cores.

Rather than parallelizing CDCL, more successful approaches run several state-of-the-art sequential SAT solvers concurrently. Even the first parallel SAT procedure mentioned above reports occasional super-linear *speedup* on satisfiable instances. Concerning unsatisfiable instances, in [CW03] super-linear *speedups* are reported for both satisfiable and unsatisfiable formulae when *clause sharing* feature is enabled. This thesis is a work on clause sharing.

Clause sharing is one of the *seven challenges for parallel SAT solving* [HW12], and it is a very actual research subject. For example, at the *Learning and Intelligent Optimisation*[2] conference of the current (at the time of writing) year (LION 7), out of 4 talks about SAT solvers, 3 of them were focusing on clause sharing.

Clause sharing talks about sending information learnt during the search process to other SAT solvers running in parallel, in order to enforce the cooperation and achieve better results. The idea of clause sharing is somehow a "semantic consequence" of the introduction of sequential CDCL solvers, as early CDCL parallelizations like GridSAT [CW03] or PaSAT [SBK01] were already exploiting this strategy. However, the effect of clause sharing on the search is not clear, and no studies beyond empirical evaluations have been published.

A clever, and thus effective, clause sharing pones a number of classical challenges in parallel computing because of communication costs. For example, when a large number of clients are sharing even a small number of clauses the total communication overhead becomes significant. This pones a limit to the scalability of such procedures, as when resources (e.g., more cores) are added improvements in run-time are not guaranteed. In order to alleviate this problem, a natural solution is to apply a filter on the clauses that can be sent. The most common approach is to put a limit on the size of the clauses that can be shared, e.g. ManySAT [HJS09b] limit allows to be shared only clauses of size up to 8. This is kind of reasonable, since shorter clauses can prune the search space more (the pruning that can be obtained from a learnt clause is inversely proportional to the clause size). However, results in recent SAT competitions [JLBRS12] unveiled how parallel solvers which do not exploit clause sharing at all, like ppfolio, or that limit clause sharing to the extreme, like plingeling, perform surprisingly well (plingeling is the winner of SAT11 competition [KSMS11], and ppfolio a simple script that runs many sequential solvers at the same time). From this one can conclude that state-of-the-art clause sharing can further be improved, and a step forward from size-based clause sharing has recently been proposed in [AHJ$^+$12a], leading to a solver that imposed itself in the last SAT Challenge [JLBRS12].

[2] http://www.intelligent-optimization.org/LION7/

Most relevant parallel SAT solvers can be divided in two families: *portfolio solvers*, where several sequential solvers compete each other over the same formula, and *search space partitioning* (also known as *divide-and-conquer*), where each solver is assigned a partition of the original problem and partitions are assigned to sequential solvers running in parallel. Despite its importance in the past, research on divide-and-conquer approach stopped in 2007 with the last divide-and-conquer solver PMinisat [GSH08], and from 2008 the state of the art is represented by portfolio solvers [HJS09b]. One reason why divide-and-conquer solvers cannot match the performance of portfolio procedures is because, in order to partition the search space in a "balanced" way (rephrased, in order to create partitions that are equally difficult), a work balancer has to be included and this has a cost. Moreover, Hyvärinen et al. [HM12a] proved that there is also a theoretical *slowdown* when the formula is partitioned.

In the same work it has been observed, however, that also portfolio solvers have drawbacks, as for example they seem not to scale well when more resources (e.g., more cores) are added (although good results have recently been achieved up to 32 cores in [AHJ$^+$12a]). Thus, they propose a hybrid approach to overcome limitations of both, called *iterative-partitioning*. Iterative-partitioning differs from classical divide and conquer in the fact that the search space is partitioned iteratively, allowing more solvers to look at the same part of the search space at the same time. Also, it significantly differs from portfolio since the search space is partitioned.

Due to its novelty, iterative-partitioning solvers have received little attention and only a limited form of clause-sharing has been proposed in [HJN11].

In this thesis a new clause sharing mechanism for the iterative-partitioning approach is proposed. This proposal is an extension of the work in [HJN11] which allows to largely increase the total number of clauses that can be shared. Furthermore, this work provides a correctness proof and an evaluation that reveals interesting insights. First, improved clause sharing does not introduce significant overhead to computation. Furthermore, the overall performance of the search is increased. This is an effect of the improved cooperation between solvers that the new technique guarantees. Finally, the approach scales with more cores; when the number of cores is increased from 4 to 16, the overall performance of the system improves.

The document is structured as follows: *Section* 2 provides basic notions and fixes the notation that will be used throughout the work. *Section* 3 describes CDCL SAT solvers in high detail, and major improvements behind modern CDCL SAT solvers, that are relevant for our work, are discussed. As parallel SAT solving is discussed in *Section* 4, search space partitioning is formally defined. Furthermore, benefits and drawbacks of existing clause sharing strategies are discussed, in order to finally focus on existing approaches to clause sharing for search space partitioning solvers. Moving from the drawbacks of these approaches, the new clause sharing technique is introduced, and it is shown how it does not suffer of the problems of other approaches. The section terminates with a correctness proof of the newly introduced clause sharing approach. An intense evaluation of the new approach is performed in *Section* 5, whereas *Section* 6 is for conclusions and possible extensions.

2 Basics

In this section the reader is introduced to notions and notation that will be used throughout the rest of the document.

Definition 2.1. A set of *atomic propositions* $\mathcal{AP}$ is a countably infinite set of symbols. In the context of SAT solving, and in this work, $\mathcal{AP} = \mathbb{N}$. Atomic propositions are also called *atoms* or *propositional variables*.

Definition 2.2. A *literal* L is a propositional variable A or its negation $\neg A$.

Definition 2.3. Given a literal L, its *complement* $\overline{L}$ is defined as:

$$\overline{L} := \begin{cases} \neg A & \text{, if } L = A \in \mathcal{AP} \\ A & \text{, if } L = \neg A, \text{ where } A \in \mathcal{AP} \end{cases}$$

Given a set of literals $\mathcal{L}$, with $\overline{\mathcal{L}}$ we denote the set containing all the complements of the literals in $\mathcal{L}$, i.e. $\{\overline{L} \mid L \in \overline{\mathcal{L}}\}$.

Definition 2.4. A *clause* is a finite set of literals.

A *unary clause* is a clause C such that $|C| = \neg 1$. A clause is a *tautology* iff it contains a complementary pair of literals.

Definition 2.5. A *clause set* is a finite set of clauses.

The clause resulting from adding a literal L to a clause C, i.e. $C \cup \{L\}$, is denoted with C, L. Likewise, F, C denotes the clause set obtained by adding the clause C to the clause set F.

Clause sets can also be understood as propositional formulae in conjunctive normal form (CNF), where each clause is a generalized disjunction of literals. Notice, however, that there is not an exact mapping between formulae and clause sets, since a formula can contain repetitions of the same element and a clause set cannot. However, duplicates can be safely removed from a formula without changing its semantics. Same holds for clauses, that is, they can be deprived from tautologies. Moreover, given a CNF formula it is straightforward to derive its corresponding clause set, and vice-versa given a clause set it is clear how to derive one of its derived formulae. For these reasons, from now on we will use the word formula in place of clause sets, and the concepts themselves of formulae in CNF and clause sets will be used interchangeably.

Given a set of propositional variables $\mathcal{A} \subseteq \mathcal{AP}$, we can define the set of all formulae over the propositional variables of $\mathcal{A}$ as $\mathcal{F}_{\mathcal{A}}$. In the following, whenever $\mathcal{A}$ is clear from the context we will write $\mathcal{F}$ in place of $\mathcal{F}_{\mathcal{A}}$. In a similar way, we define the set of all clauses as $\mathcal{CLS}_{\mathcal{A}}$ or $\mathcal{CLS}$.

Definition 2.6. The function $atom : \mathcal{AP} \cup \overline{\mathcal{AP}} \to \mathcal{AP}$ associates to a literal L a set containing its underlying propositional variable, i.e.

$$atom(L) = \begin{cases} \{A\} & \text{, if } L = A \in AP \\ \{A\} & \text{, if } L = \neg A, A \in \mathcal{AP} \end{cases}$$

Similarly, the function $atom : \mathcal{CLS} \rightarrow \mathcal{AP}$ associates to a clause the set of propositional variables in that clause:

$$atom(C) = \begin{cases} \{\} & \text{, if } C = \{\} \\ atom(C') \cup atom(L) & \text{, if } C = C', L \end{cases}$$

Finally, $atom : \mathcal{F} \rightarrow \mathcal{AP}$ associates to a formula F the set of the propositional variables elements of some clause in F:

$$atom(F) = \begin{cases} \{\} & \text{, if } F = \{\} \\ atom(F') \cup atom(C) & \text{, if } F = F', C \end{cases}$$

Definition 2.7. An *interpretation* is a (partial or total) assignment from the set of propositional variables $\mathcal{AP}$ to the set of truth values $\{\top, \bot\}$.

A finite interpretation I can be represented by a sequence of literals where no duplicates can appear in the sequence and the sequence does not contain any complementary pair. We extend the sequence J with a literal L by adding it to its right side, and we write J, L. Similarly, the sequence result of the concatenation of two sequences J and J' is denoted by J, J'. The set of all the elements in the sequence M is denoted as $s(M)$. Given a sequence of literals M (devoid of duplicates and complementary pairs), the corresponding interpretation I is defined as :

$$I(L) := \begin{cases} \top & \text{, if } L \in s(M) \\ \bot & \text{, if } \neg L \in s(M) \end{cases}$$

From now on, whenever the word "interpretation" is used, it is meant the corresponding sequence of literals.

Definition 2.8. Given an interpretation J, the *complement* $\overline{J}$ is the interpretation obtained by complementing all the literals in J.

Definition 2.9. A clause C is *satisfied* by an interpretation J, in symbols $J \models C$, if there is a literal L in C s.t.

$$J(L) = \top$$

Definition 2.10. A formula F is *satisfied* by an interpretation J, in symbols $J \models F$, if all of its clauses are satisfied

A central notion in SAT solving is the one of *reduct*:

Definition 2.11. Given a formula F and an interpretation J, the *reduct* $F|_J$ of F w.r.t. J is the formula obtained from F by removing every satisfied clause and every literal L s.t. $J(L) = \bot$.

The next example clarifies the notion of reduct for a formula:

Example 2.12. Let $F := \{\{1, -2\}, \{3\}, \{4, -5\}\}$, and $J := (1, -3, 5)$. Then

$$F|_J := \{\{\ \}, \{4\}\}$$

An central notion in logic is the one of *semantic consequence*:

Definition 2.13. Let F, G be formulae. Then G is a *semantic consequence* of F, denoted as $F \models G$, iff for every interpretation I the following holds:

$$I \models F \Rightarrow I \models G$$

The previous definition is important to define a central notion in propositional logic (and in SAT solving as well), that is the notion of *semantic equivalence*:

Definition 2.14. Let F, G be formulae. Then F and G are *semantically equivalent*, denoted as $F \equiv G$, iff:

$$F \models G \text{ and } G \models F$$

A clause C is a semantic consequence of a formula F iff $F \models \{C\}$. A common methodology to derive implied clauses (alias, semantic consequences) from a formula is by means of the *resolution rule*:

Definition 2.15. Let C, D be clauses, and consider a literal L in C s.t. $\overline{L}$ occurs in D. Then the *resolvent* $C \otimes D$ of C and D over the literal L is the clause $(C \setminus L) \cup (D \setminus \overline{L})$.

Definition 2.16. Let $F := \{C_1, \ldots, C_n\}$ be a formula. Then a *resolution derivation* for F can be defined by structural induction:

1. the sequence $(C_1, \ldots, C_n)$ is a resolution derivation for F
2. if $(R_1, \ldots, R_m)$ is a resolution derivation for F, then the sequence of clauses $(R_1, \ldots, R_m, R_{m+1})$, where $R_{m+1} := R_i \otimes R_j$ and $i, j \in \{1, \ldots, m\}$, is a resolution derivation for F

3 Sequential SAT Solving

In complexity theory, SAT is a hard problem. Of course this does not imply that, in practice, "efficient" procedures to solve SAT, at least for certain instances, cannot be found. However, for over 40 years SAT remained an intractable problem, even in practice.

Nowadays things substantially changed, and modern SAT Solvers are able to effectively tackle problems ranging over a wide and heterogeneous set of fields like planning [KS92], scheduling [HMS10, CP89], vehicle routing [Goe10], hardware and software verification [BCCZ99], [DKW08], and many others. The major improvement to the canonical DPLL, whose (many) modern variants are currently the best SAT solvers available for industrial instances, has been achieved with the introduction of the *Conflict Driven Clause Learning* technique, implemented for the first time in the SAT solver called GRASP [SS96]. DPLL solvers with clause learning technique are usually referred to as CDCL solvers. Katebi et al. show in [KSMS11] that from all the major improvements to SAT solvers, clause learning is the most beneficial technique. That contribution was so important that nowadays CDCL solvers are the best sequential algorithm for solving the SAT problem in a wide range of instances. However, although important, that contribution alone was still not sufficient to make SAT solving appealing in practice. The impressive performance of modern CDCL procedures, indeed, is the result of several small improvements which optimise to the maximum extent the usage of the underlying hardware architecture. A main example of this fact is the *two-watched-literal* scheme for unit propagation, first proposed in solver CHAFF [MMZ$^+$01] in 2001. Furthermore, the introduction of well-tuned heuristics, like for example VSIDS heuristic for decision variable selection [MMZ$^+$01] or luby heuristic for restart policy [LSZ93], has dramatically improved the performance. To confirm this, a recent research of Frank Hutter, Holger Hoos et. al. [HHLB13] (in proceedings of publication) on parameter tuning on three relevant $\mathcal{NP}$ problems shows how, among the parameters available for a modern SAT solver, the most crucial one ("by a wide margin") is the choice of the heuristic for performing decision variable selection.

This section will describe sequential CDCL SAT solvers in high detail. First, it introduces the key elements of a general CDCL procedure by means of an Abstract Reduction System (ARS) [BN98]. The ARS here proposed is based on the rules of the General DPLL Calculus in [Arn09]. *Section* 3.3.1, instead, explains in detail how the conflict analysis for deriving learnt clauses is performed in modern CDCL solvers. We conclude this section by discussing other crucial improvements of CDCL procedures, namely *two-watched* literal scheme for unit propagation, decision heuristics, restart policies and management of learnt clauses database.

3.1 Abstract Reduction Systems

The aim of this subsection is to give an introduction to Abstract Reduction Systems and their basic properties. Although the notions here given are more than enough for a complete description of the DPLL calculus, the interested reader is remanded

to [BN98] for a complete survey of the subject.

Definition 3.1. Let A be a set, and $\to$ a binary relation on A. Then, the pair $(A, \to)$ is called an *abstract reduction system* (ARS).

We write $x \to y$ instead of $(x, y) \in \to$. Relation $\to$ is known in literature as *reduction*. A chain $x_1 \to x_2 \to \dots$ is called *reduction chain*.

Definition 3.2. Let $(A, \to)$ be an ARS. Then we define the following relations:

1. $\overset{0}{\to} := \{(x, x) \mid x \in A\}$, i.e. reflexive closure of $\to$
2. $\overset{i+1}{\to} := \{(x, y) \mid \exists z \in A.\ x \overset{i}{\to} z \text{ and } z \to y\}$
3. $\overset{*}{\to} := \bigcup_{i \geq 0} \overset{i}{\to}$, i.e. reflexive and transitive closure of $\to$
4. $\overset{+}{\to} := \bigcup_{i > 0} \overset{i}{\to}$, i.e. transitive closure of $\to$
5. $\leftarrow := \to^{-1}$, i.e. inverse relation

Definition 3.3. Let $\mathcal{A} := (A, \to)$ be an ARS. Then $x, y \in A$ are said to be *joinable*, denoted as $x \downarrow y$, iff

$$\exists z \in A.\ x \overset{*}{\to} z \overset{*}{\leftarrow} y$$

Definition 3.4. The reduction $\to$ is *confluent* iff

$$y_1 \overset{*}{\leftarrow} x \overset{*}{\to} y_2 \Rightarrow y_1 \downarrow y_2$$

Definition 3.5. The reduction $\to$ is *terminating* iff there is not an infinite chain $x_0 \to x_1 \to x_3 \to \dots$

Definition 3.6. The reduction $\to$ is said to be *locally confluent* iff

$$y_1 \leftarrow x \to y_2 \Rightarrow y_1 \downarrow y_2$$

Lemma 3.7. *If $\to$ is terminating, then*

$$\textit{locally confluence} \Leftrightarrow \textit{confluence}$$

We say that an ARS $(A, \to)$ is confluent (terminating, locally confluent) iff $\to$ is confluent (terminating, locally confluent).

3.2 $\mathcal{CDCL}$ Abstract Reduction System

Remark. For this section we assume that every literal in a partial assignment J can be either a *decision* literal $\dot{L}$ (denoted with a "dot" as a superscript) or a *propagation* literal L. As a convention, we denote a list of propagation literals with the upper-case letter P.

Table 1: $\mathcal{CDCL}$ Abstract Reduction System

(1)	F	::	J	$\rightsquigarrow_{sat}$	SAT			iff $F\|_J = \{\}$
(2)	F	::	J	$\rightsquigarrow_{unsat}$	$UNSAT$			iff $\{\} \in F\|_J$ and $level(J) = 0$
(3)	F	::	J	$\rightsquigarrow_{dec}$	F	::	$J, \dot{L}$	iff $L \in \mathsf{atoms}(F) \cup \overline{\mathsf{atoms}(F)}$ and $L \notin J$ and $\neg L \notin J$
(4)	F	::	J	$\rightsquigarrow_{unit}$	F	::	J, L	iff $[L] \in F\|_J$
(5)	F	::	J	$\rightsquigarrow_{learn}$	F, C	::	J	iff $F \models C$ and $C \subseteq \text{lit}(F)$
(6)	F	::	$J', \dot{L}, J$	$\rightsquigarrow_{back}$	F	::	J', L'	iff $[L'] \in F\|_{J'}$
(7)	F	::	$P, \dot{L}, J$	$\rightsquigarrow_{restart}$	F	::	P	

Definition 3.8. Consider the sets CLS and $PASS$ of clause sets and partial assignments over a set of propositional variables $\mathcal{AP}$. Then we define the set of nodes $Nodes$ of $\mathcal{CDCL}$ as:

$$Nodes \subseteq (CLS \times PASS) \cup \{SAT, UNSAT\}$$

For aesthetic reasons, we denote every node of the kind $(F, J) \in Nodes$ as $F :: J$.

Definition 3.9. A *rule* is a reduction $\rightsquigarrow_R \subseteq Nodes \times Nodes$. Rules of $\mathcal{CDCL}$ are presented in *Table* 1.

Definition 3.10. Consider a set $Rules := \{sat, unsat, dec, unit, back, learn\}$ containing the names of $\mathcal{CDCL}$ rules. Then we define the abstract reduction system $\mathcal{CDCL} := (Nodes, \rightsquigarrow)$ as the ARS where

$$\rightsquigarrow := \bigcup_{R \in Rules} \rightsquigarrow_R$$

The rules of the $\mathcal{CDCL}$ abstract reduction system are depicted in *Table* 1, where with $level(J)$ it is meant the number of decision literals in J. The rules aim to construct a model for the formula. If such a model is found, then the algorithm stops in the irreducible node SAT by an application of rule (1).

The second rule is applied only once the algorithm has reached a conflict and no more backtrack points (alias, decision literals) are available.

The third rule, that in the following we will call *decision rule*, guides the search by (heuristically) choosing an unassigned variable and assigning it. Since this choice might be not the right choice for finding a model for the formula, provided that the formula is satisfiable, the information that this is a candidate backtrack point has to be kept (we do it by superscripting the choice with a dot).

The fourth rule, known in literature as *unit propagation*, performs deduction in the sense that it identifies those literals that have to necessarily be added to the current partial assignment in order to construct a model the formula.

Learn rule permits to add clauses to the formula without changing the underlying satisfiability problem (i.e., without changing the search space for the problem), and it is usually used in combination with the *back* rule in order to allow the solver to

escape from the current part of the search by "backtracking" to one of the previously saved backtrack points.

The last rule, *restart*, restarts the whole search process from scratch, but keeps all those literals found by unit propagation steps performed before the first decision.

A sequence of rule applications starting from the empty assignment is here called a *run*.

Definition 3.11. An *run* of $\mathcal{CDCL}$ over the formula F is a reduction chain starting from the node $F :: (\ )$ over the $\mathcal{CDCL}$ abstract reduction system $(Nodes, \rightsquigarrow)$, where $Nodes$ is defined by assuming $\mathcal{AP} = atom(F)$.

The *application* of a rule R over a couple of nodes n_1, n_2 is the reduction step $n_1 \rightsquigarrow_R n_2$. Observe that an algorithm for solving the SAT problem can be obtained from the $\mathcal{CDCL}$ abstract reduction system by choosing a certain rule application strategy. For example, a common strategy to every modern SAT solver prefers unit-propagation rule applications against over other applicable rules. From now on we will assume that every run follows this strategy.

The $\mathcal{CDCL}$ calculus is sound and complete, i.e. given a formula F, every possible run for F leads to a node SAT ($UNSAT$) iff F is satisfiable (unsatisfiable). Under the assumption that no run is allowed to contain an infinite number of *restart* rule applications (*fairness* constraint), termination can be proved as well [Arn09].

$\mathcal{CDCL}$ system is a very powerful tool to describe key concepts of CDCL solvers, as it provides a formal base necessary to prove important results. Moreover, $\mathcal{CDCL}$ reduction chains offer a convenient notation to describe examples of execution of practically every CDCL procedure. However, being a high-level formalisation that tries to catch keys features common to every CDCL solver, it suffers from lack of details. For example, how do modern SAT solvers find semantic consequences that can be learnt? Also, what literals should be chosen by the decide rule? Answers to these questions are what really makes the difference between a 40 years old algorithm and modern competitive real world CDCL solver. Of course, going through all these details is not the goal of this work. Still, some details are necessary in order to continue our discussion and are treated in next section.

3.3 Modern CDCL Solvers

3.3.1 Clause Learning

In modern sat solvers, the semantic consequences necessary for the application of the *learn* rule are found by using *conflict graphs* [SS96]. Conflict graphs are one of those (few) key concepts in modern SAT solving that has been properly formalised. The formalisation here presented complies with the one in the SAT lectures of Prof. Hölldobler (year 2011).

Definition 3.12. Let $F :: J$ be a node of the $\mathcal{CDCL}$ abstract reduction system, and consider a literal L s.t. $J = I, L, I'$. Then the level $level(atom(L))$ of the propositional variable $atom(L)$ under the interpretation J is the number of decision literals in I, L.

In order to derive the learnt clause, a study on the clauses satisfied (or falsified) by unit propagations has to be performed. The set of these clauses w.r.t. a node $F :: J$ and the propagated literal L is denoted as $relevantL(F :: J)$, and formally defined in *Definition* 3.13.

Definition 3.13. Consider a run $F :: () \overset{*}{\rightsquigarrow} F' :: J$ of the $\mathcal{CDCL}$ reduction system over the propositional formula F. Then the *set of relevant clauses for* L w.r.t. the node $F' :: J$, denoted as $relevantL(F' :: J)$, is defined as:

$$relevantL(F' :: J) := \{C \in F' |\ \exists I.\ I, L, I' = J \text{ and } C|_I = \{L\}\}$$

In the following we assume that every set $relevantL(F :: J)$ has cardinality one. In order to enforce this one can partition a set $relevantL(F :: J)$ into several unary sets. Each of these sets will lead to a different *implication graph*.

Definition 3.14. Consider a run $F :: () \overset{*}{\rightsquigarrow} F' :: J$ of the $\mathcal{CDCL}$ reduction system over the propositional formula F. Then an *implication graph* for $F' :: J$ is a pair $(\mathcal{V}, \mathcal{E})$ where:

1. $\mathcal{V} := J$
2. $\mathcal{E} := \{(L, L') |\ L, L' \in \mathcal{V}, relevantL(L', F :: J) = \{C\}, \overline{L} \in C\}$

Now we have all the ingredients to formally define the notion of *conflict graph*:

Definition 3.15. Consider a run $F :: () \overset{*}{\rightsquigarrow} F' :: J, P$ of the $\mathcal{CDCL}$ reduction system over the formula F such that $C := \{\} \in F'$. Then a *conflict graph* $(\mathcal{V}', \mathcal{E}')$ is a directed graph such that:

1. $\mathcal{V}' := V \cup \{k\}$
2. $\mathcal{E}' := \mathcal{E} \cup \{(L, K) |\ L \in \mathcal{V} \text{ and } \overline{L} \in C\}$

Given a conflict graph, we associate a labelling function to its nodes and its arches in the following way: each literal node L is labelled with $level(atom(L))$, and each arch $(L, \overline{L})$ is labelled with the clause C that produced the arch (in MiniSat [ES03], this clause is known as the *reason* for the propagation). Next example displays how to construct a conflict graph given a node in a $\mathcal{CDCL}$ run for a given propositional formula.

Example 3.16. This example is inspired by [MSLM]. Consider the formula

$$F := \{\{1, 2, \neg 3\} := C_1, \{1, \neg 4\}, \{3, 4, 5\}, \{\neg 5, \neg 6\}, \{7, \neg 5, \neg 8\}, \{6, 8\} := C_6\}$$

and the following $\mathcal{CDCL}$ run:

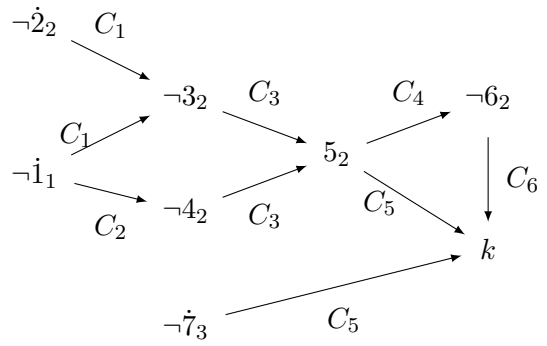

Figure 1: Conflict graph for the node of *Example* 3.16. Literal nodes are subscripted with their levels, and each arch is labelled by the clause that generated it.

$$
\begin{array}{ll}
 & F :: () \\
\rightsquigarrow_{dec} & F :: (\neg\dot{2}) \\
\rightsquigarrow_{dec} & F :: (\neg\dot{2}, \neg\dot{1}) \\
\rightsquigarrow_{unit} & F :: (\neg\dot{2}, \neg\dot{1}, \neg 4) \\
\rightsquigarrow_{unit} & F :: (\neg\dot{2}, \neg\dot{1}, \neg 4, \neg 3) \\
\rightsquigarrow_{unit} & F :: (\neg\dot{2}, \neg\dot{1}, \neg 4, \neg 3, 5) \\
\rightsquigarrow_{unit} & F :: (\neg\dot{2}, \neg\dot{1}, \neg 4, \neg 3, 5, \neg 6) \\
\rightsquigarrow_{dec} & F :: (\neg\dot{2}, \neg\dot{1}, \neg 4, \neg 3, 5, \neg 6, \neg\dot{7}) \\
\rightsquigarrow_{unit} & F :: J := (\neg\dot{2}, \neg\dot{1}, \neg 4, \neg 3, 5, \neg 6, \neg\dot{7}, \neg 8)
\end{array}
$$

Observe that $\{6, 8\}|_J = \{\}$. A possible conflict graph for the node $F :: J$ is the one in *Figure* 3.16.

For conflict analysis it is important to consider only the maximal connected subgraph containing the node k. Thus, from this point onwards with the notion "conflict graph" we will refer to this component, only. In order to derive a learnt clause from a conflict graph, we need the notions of *paths* and *cuts*.

Definition 3.17. Let $G := (V, E)$ be an oriented graph. Then a *path* $p := p_1 p_2 \dots p_n$ in G is a sequence of nodes such that, for each $1 \leq i < n$, $(p_i, p_{i+1}) \in E$.

Definition 3.18. Let $(\mathcal{V}, \mathcal{E})$ be a conflict graph. Consider a subset $\mathcal{V}_\mathcal{S} \subseteq \mathcal{V}$ containing all the decision literals and the literals of level zero in $\mathcal{V}$. Then the set $paths(\mathcal{V}, \mathcal{E})$ denotes the set of all those paths from the nodes in V_S to the node k.

Definition 3.19. Consider a conflict graph $(\mathcal{V}, \mathcal{E})$. Let $\mathcal{V}_\mathcal{S}$ be the subset of $\mathcal{V}$ containing all the decision literals and the literals of level zero in $\mathcal{V}$. Then a *cut* $(\mathcal{V}_\mathcal{R}, \mathcal{V}_\mathcal{C})$ is a partition of $\mathcal{V}$ into $\mathcal{V}_\mathcal{R}$, $\mathcal{V}_\mathcal{C}$ such that:

- $\mathcal{V}_\mathcal{S} \subseteq \mathcal{V}_\mathcal{R}$
- $\{0\} \subseteq \mathcal{V}_\mathcal{C}$
- each path $p \in paths(\mathcal{V}, \mathcal{E})$ can be partitioned into two sub-paths p_R and p_C such that:
  - $p = p_R p_C$
  - $\mathcal{V}_\mathcal{R}$ contains all the nodes in p_R
  - $\mathcal{V}_\mathcal{C}$ contains all the nodes in p_C

Each cut corresponds to a clause that can be learnt from the sequential solver.

Definition 3.20. Let $(\mathcal{V}_\mathcal{R}, \mathcal{V}_\mathcal{C})$ be a cut for a conflict graph $(\mathcal{V}, \mathcal{E})$. Consider the sub-graph $(\mathcal{V}_\mathcal{R}, \mathcal{E}_\mathcal{R})$ induced by the set of nodes $\mathcal{V}_\mathcal{R}$ on $(\mathcal{V}, \mathcal{E})$. Consider the set of nodes $V'_R \subseteq \mathcal{V}_\mathcal{R}$ that do not have a successor in $(\mathcal{V}_\mathcal{R}, \mathcal{E}_\mathcal{R})$. Then the set of literals $\overline{\mathcal{V}'_\mathcal{R}}$ is called the *cut clause* determined from the cut $(\mathcal{V}_\mathcal{R}, \mathcal{V}_\mathcal{C})$ over the conflict graph $(\mathcal{V}, \mathcal{E})$.

Given a conflict graph $\mathcal{G}$ for a node $F :: J$, it can be proved that every cut clause originated from $\mathcal{G}$ is a semantic consequence of the formula F. This is proved by showing that each cut clause can be obtained by means of a (linear) resolution derivation over the reason clauses and that start from the conflict clause. Note that the other direction does not hold, i.e. given a linear resolution derivation over the reason clauses starting from the conflict clause, the result of the derivation could be not a cut clause [PD10]. *Example* 3.21 shows how to derive a cut clause by means of resolution steps.

Example 3.21. Consider again the node $F :: J$ in *Example* 3.16, and the conflict graph in *Figure* 3.3.1 with the cut clause $\{1, 2, 7\}$ in evidence. Then the same cut clause can also be obtained by means of the following linear resolution derivation over the reason clauses $C_1, \ldots, C_6$ that starts from the conflict clause $\{6, 8\}$:

C_1	$\{1, 2, \neg 3\}$	relevant clause
C_2	$\{1, \neg 4\}$	relevant clause
C_3	$\{3, 4, 5\}$	relevant clause
C_4	$\{\neg 5, \neg 6\}$	relevant clause
C_5	$\{7, \neg 5, \neg 8\}$	relevant clause
C_6	$\{6, 8\}$	relevant clause
C_7	$\{7, \neg 5, 6\}$	$C_6 \otimes C_5$
C_8	$\{7, \neg 5\}$	$C_7 \otimes C_4$
C_9	$\{3, 4, 7\}$	$C_8 \otimes C_3$
C_{10}	$\{3, 1, 7\}$	$C_9 \otimes C_2$
C_{11}	$\{1, 2, 7\}$	$C_{10} \otimes C_1$

The resolution procedure that is shown in *Example* 3.16 is the kind of conflict analysis that is performed in modern implementations of sequential SAT solvers. Since

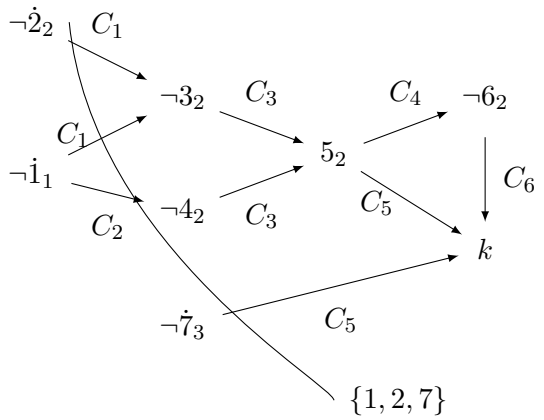

Figure 2: Conflict graph for the node of *Example* 3.16, and the cut clause $\{1, 2, 7\}$

every cut is a potential learnt clause, being each cut a semantic consequence of the original formula (this can be proved by induction), a SAT solver needs a policy on when to stop resolution steps and consequently on which cut has to be chosen. The most common strategy called the *first unique implication point* technique [ZMM01], that we will not discuss here.

3.3.2 VSIDS Decision Heuristic

Probably the most crucial part of every search procedure is the choice of the decision heuristic. SAT solving makes no exception, as recent work of Frank Hutter, Holger Hoos et. al. [HHLB13] displays how, among all of the parameters available in a modern SAT solver, the choice of the right decision heuristic is "by a wide margin" the one that has the highest impact on the overall performance. In our setting, a decision heuristic would determine which new variable and polarity should be selected each time $\rightsquigarrow_{dec}$ is applied. At the time of writing, the most popular decision heuristic for CDCL solvers is *Variable State Independent Decay Sum* (VSIDS) [MMZ$^+$01], used in the majority of state-of-the-art SAT solvers like MiniSat [ES03].

VSIDS heuristic has been introduced with the CDCL solver Chaff [MMZ$^+$01], and at the time of its introduction authors reported an improvement of one order of magnitude over most difficult instances, without affecting the performance over easy ones. In a CDCL solver with VSIDS decision heuristic, every literal L is associated an activity $activity(L)$. This activity is initialised to zero for each literal, or to a random value. Each time a clause C is learnt, $activity(L)$ is incremented of a certain value inc, for every literal L occurring in C. At each decision rule application, the unassigned variable with the highest activity is chosen. In order to efficiently keep track of the unassigned variable with highest activity, MiniSat 2.2 makes use of an ordered

heap data structure. Periodically, each activity is decreased by a constant *decay*. Observe that decaying the activities for each variable can be done in constant time by simply incrementing the increment value *inc*. In MiniSat 2.2, decaying of variables is performed at each conflict by incrementing the value *inc* as $inc := inc \times decay$.

Clearly, the aim of VSIDS heuristic is to satisfy most *recent* conflict clauses. As learnt clauses primarily drive the search process on difficult instances (because of the increased number of conflicts), VSIDS enforces this driving process. Observe that, since at each conflict the activity update takes place only for a very limited number of literals (namely, the ones in the learnt clause), keeping track of activities is very cheap and does not require any complex analysis on the structure of the problem.

The VSIDS strategy used by MiniSat is a bit different than the standard one used by Chaff. First, an activity is kept for every propositional variable instead of every literal. Secondly, activities are incremented not only for the literals in the learnt clause but for every variable in every clause used in the resolution derivation of the learnt clause. Moreover, decaying is performed at every conflict (of a small value). [ES05] reports that the new scheme is able to drive the search towards the satisfaction of learnt clause more quickly than the original VSIDS method.

3.3.3 Two-watched Literals Unit Propagation

Work in [Man11] reports how, in real world modern SAT solvers, around 80% of the total run-time is spent in unit-propagation steps (although two-watched literals scheme is used). Therefore, an efficient unit-propagation engine is key to any SAT solver.

Two-watched literals scheme has been introduced in the solver Chaff [MMZ$^+$01] and consists of a low-overhead technique for keeping track of those clauses which can trigger unit-propagation steps.

Together with VSIDS decision heuristic, two-watched literals scheme allowed Chaff to obtain one to two orders of magnitude performance improvement on difficult SAT instances in comparison with other DPLL or CDCL solvers of its time, like SATO [Zha97] or Grasp [SS96].

For each clause C, two unassigned literals are chosen to be *watched*. We denote the (binary) set of literals watched by a clause C as $watchedBy(C)$. Accordingly, for each literal L we define the set $watchersOf(L) := \{C \mid L \in watchedBy(C)\}$ as the set of clauses watching L. Whenever a literal L is assigned to false, corresponding watcher clauses $watchersOf(L)$ are searched for a new unassigned literal to watch. If for some clause $C \in watchersOf(L)$ this choice is not possible, then this means that every literal in C but one is assigned and a unit-propagation step can take place. Of course, the only unassigned literal in C is the literal $M \in watchedBy(C)$ such that $M \neq L$.

3.3.4 Learnt Clauses Database Management

Unit-propagation in modern SAT solvers requires to keep track of two literals for every clause in the original formula and for every learnt clause (see *Section* 3.3.3).

Therefore, the size of the database storing learnt clauses has a direct impact on the performance of unit-propagation steps. Since an efficient unit-propagation engine is crucial to the performance of every SAT solver, learnt clauses database should always be kept to a feasible size; for this purpose, a learnt clauses deletion policy able to remove *unnecessary* clauses from the learnt database is required.

Unfortunately, determining what learnt clauses are unnecessary at a certain point of execution is a hard task, and thus deletion strategies have to rely on robust heuristics. MiniSat 2.2 uses an approach very similar to VSIDS heuristic, called *Activity Based Clause Removal* [ES03]. Activities are kept for each clause, and activity for a clause increased each time this clause participates as a resolvent to the resolution derivation of a learnt clause. Decaying of clause activities comply with the variables decaying policy of VSIDS decision heuristic.

A more recent work [AS09], instead, uses a more aggressive cleaning strategy because of a better estimation of clause usefulness. This is the strategy used by the solver glucose [AS12], winner of the last SAT Challenge 2012. The usefulness of each clause is defined in terms of *Literal Block Distance* (lbd) measure:

Definition 3.22. Consider a run $F :: () \overset{*}{\rightsquigarrow} F', C :: J$ for a formula F in the $\mathcal{CDCL}$ abstract reduction system. Consider the set $levels(C)$ of all the levels of the literals in C, defined as $levels(C) := \{n \mid level(L) = n, L \in C\}$. Then $lbd(C) := |levels(C)|$ denotes the *lbd measure* of the clause C w.r.t. the current partial assignment J.

Using the same terminology as [AS09], a *block* of literals is a sequence of literals having the same level. Intuitively, learning a clause C such that $lbd(C) = 2$ means that every literal of the last decision level will be *glued* in the current assignment to the block of literals of the backtrack level, no matter what the size of the clause is.

Cleaning strategy in glucose removes, every a certain number of conflicts, all those non-binary clauses with lbd greater than 2 belonging to the first half of the learnt clauses database (the oldest ones). Experimental results show effectiveness of lbd measure in evaluating learnt clauses relevance, as work in [AS09] reports an improvement of performance around an order of magnitude on both satisfiable and unsatisfiable industrial instances.

4 Parallel SAT Solving

Up to some years ago, hardware improvements were mainly achieved by improvements of the CPU clock speed. In this scenario, parallel architectures were not widely spread and programmers did not bother much on implementing parallel algorithms. Improvements on clock speed, indeed, always means an improvement on the execution time of every sequential procedure.

Unfortunately, the situation changed when CPU manufacturers hit the thermal wall, in around 2004. At that point, in order to improve performance, only an option was left: pack more cores in the same CPU. In other words, the era of parallel architectures had finally arrived.

Though we said that the thermal clock was hit no more than ten years ago, research on parallel algorithms has deep roots, and many "big names" in computer science community challenged themselves with the problems that parallelism pones; for example, "semaphores" were invented in 1965 by Edsger W. Dijkstra [Dij68], or the "Dining Philosophers Problem" (in its most famous formulation) was introduced from the "quick-sort" inventor, C.A.R. Hoare, in [Hoa78]. Parallelism is fascinating, due to its double-sided nature of chance and challenge: parallelism is a chance, since many algorithms may benefit from exploiting the usage of parallel resources; on the other hand, parallelism pones several challenges and potential benefits may be nullified if these problematics are not taken into account properly.

This section is devoted to parallel SAT solving, and it is organised as follows: first it gives an overview on related work, starting from the first attempts to DPLL parallelisation until the current state-of-the-art CDCL based parallel approaches. This part of the discussion is written in spirit of work in [HMN$^+$11] and [BP10]. We then move our attention on the particular class of parallel SAT solvers known as *Iterative Partitioning* solvers, and motivate their use as a backbone for our work.

After a digression over related work on clause sharing, in *Subsection* 4.5 we thoroughly discuss the current state-of-the-art techniques for what concerns clause sharing in an Iterative Partitioning setting, exposing both the advantages and drawbacks for such techniques. Finally, *Subsection* 4.7 introduces the main contribution of this work, namely *Position-based clause tagging* clause sharing approach, and shows how the new sharing profits from the weaknesses of other approaches. This section is concluded by a formal correctness proof for the proposed approach.

4.1 Evaluation of Parallel Procedures

Parallel procedures are traditionally evaluated in terms of *speedup* and *efficiency*. Let T_s represent the *wall-clock* time used by a sequential procedure to solve a certain instance of a problem, where with *wall-clock time* it is meant the number of seconds elapsed from the start of the procedure to its termination. Likewise, let T_p be the wall-clock time that a parallel procedure spent on solving the same instance. Then the *speedup* S is the fraction $\frac{T_s}{T_p}$. If n is the number of parallel computation units allowed to the parallel procedure (e.g., the number of cores), then *efficiency* E is defined as the fraction $\frac{S}{n}$. A speedup is said to be *super-linear* when $E > 1$.

A common trait of most parallel SAT procedures is that they display occasional super-linear speedups when the input formula is satisfiable. This is somewhat not surprising, as for example a divide-and-conquer algorithm could be lucky and find a model for its partition (thus, for the original formula) by making better (and casual) choices than the original procedure. The same holds in case of portfolio procedures.

If the formula is unsatisfiable, instead, super-linear speedups are more difficult to be obtained. This is especially true for divide-and-conquer procedures, as solving a partition does not imply solving the original problem. Moreover, every divide-and-conquer procedure suffers from the flaw described in [HM12a], and even the most successful approaches like the ones mentioned in *Section* 4.2 present occasional slowdowns (only on easy instances, though). However, better (i.e., close to 1) efficiencies for unsatisfiable instances are likely to be achieved when clause sharing is enabled. Examples of solvers where these efficiencies have been reported are given in *Section* 4.5.

It is natural to expect that allowing more resources (e.g., more cores) to a parallel procedure would automatically result in improved performance. Unfortunately, this is often not true since adding more resources implies increasing the number of execution units working concurrently: if these units communicate, then communication costs could become unaffordable and worsen the overall performance.

We call *scalability* the ability of a procedure to benefit from the addition of resources. The field of SAT solving, where the testing of parallel procedures at different SAT competitions never exceeded the number of eight threads, at the best of author knowledge has not produced large scale parallel procedures yet, but it is evident that the current trend in hardware manufacturing is pushing towards such a need.

The aforementioned argument explains why in this work our group opted for an *iterative-partitioning* backbone, rather than going for a more conventional (and, currently, state-of-the-art) portfolio approach. More details for this choice can be found in *Section* 4.3. The final evaluation shows how the new clause sharing technique does not pone too strict bounds to scalability, as an improved behaviour can still be obtained when 16 cores are used.

4.2 Related Work on Parallel SAT Solving

As discussed in the introduction to this work, parallel SAT solvers can be divided, modulo a few outliers, into two main families: divide-and-conquer and portfolio. The current state-of-the-art is represented by portfolio procedures, although divide-and-conquer procedures were predominant in the past. However, portfolio is effective only in those situations where it is possible to generate a sufficient (w.r.t. the available resources) number of effective (different) solver configurations; finding such configurations is a hard task. Divide-and-conquer, instead, does not suffer of this problem: if more resources are available, then more partitions can be produced in order to assign a (meaningful) job to each computation unit.

Thus, at the current state there is not a fully dominant approach, and evidence of this is given in [MML10], where it has been shown as a combination of portfolio and divide-and-conquer (obtained by means of the shared clauses mechanism) can outperform both approaches. Therefore, it is important to study divide-and-conquer

as well as portfolio, and consequently historical details of both are evenly important.

The idea of a divide-and-conquer algorithm is to partition the search space, and to assign each partition to a sequential (state-of-the-art) sequential solver. Traditionally, the partitioning of the search space has been done with the *guiding-path* technique.

Guiding-paths, first mentioned in PSATO parallel SAT solver [ZBH96], are a easy way of keeping track of the current state of the search. A node in the guiding path is a couple (L, δ), where L is a literal and δ can be either 1 or zero. A node with $\delta = 1$ is said to be *opened*, and this means that L is a decision literal. In contrast, a node where $\delta = 0$ is said to be *closed*. A guiding-path is a (finite) sequence of such nodes.

The main idea behind guiding-path is that, whenever a computation unit finishes its job, e.g. because of a timeout, a new split is produced for a opened node in the path. In PSATO, that is a grid-based parallelisation of the DPLL algorithm, the chosen node will always correspond to the first decision literal in the current partial interpretation. Evaluation of PSATO shows a performance that "varies dramatically from case to case" when the formula is satisfiable. For unsatisfiable instances, instead, speedups are always sub-linear and efficiency becomes close to 1 only when the size of the instance increases.

One of the main problems of divide-and-conquer is the fact that it is hard to produce jobs that are equally (or almost equally) hard to solve. Unfortunately, guiding-paths approach accentuates this phenomenon. Dynamic workload balancing and work stealing techniques, introduced with the DPLL based parallel solver PSATZ [JLU01], explicitly address this problem. Whenever a computation unit becomes idle, the master runs an estimation algorithm to find out what is the "most loaded" computation unit. Then, it splits the job of that unit and assigns it to the idle units (work stealing). Evaluation reports occasional superlinear speedups for satisfiable instances, and an efficiency close to 1 when the instance is unsatisfiable.

Contemporary to the solver PSATZ is the CDCL based parallel SAT solver PaSAT [SBK01], that implements a shared-memory communication scheme and that we already mentioned as being one of the first successful attempts of implementing clause sharing among different units. Evaluation shows how clause sharing plays an important role for the performance of this algorithm as the number of observed superlinear speedups for satisfiable instances increases. Again, average efficiency on both satisfiable and unsatisfiable instances is close to 1.

The solver by Plaza et al. [PKA$^+$06], instead, is noticeable for the fact that it introduces the idea of choosing the splitting variables through VSIDS [MMZ$^+$01] heuristic; in our experiments we will use a similar strategy, with the difference that not literals, but sets of clauses will be derived and used to create partitions (see *Section* 4.4).

As multi-core architectures became available, multi-threaded solvers started to be developed accordingly. The solver ySAT [FDH05] is a multi-threaded parallel SAT solver with clause sharing, where all clauses all physically shared among the solvers. Due to the huge number of shared clauses, and to the observation that frequent cache misses occurs as the number of cores increases (situation that dramatically changed in present days, where even cheap multi-level caches can reach sizes in the order of 10^2 megabytes), the authors conclude that CDCL parallelizations on multi-core machines

cannot be performed efficiently.

One has not to wait long for a work reaching completely opposite conclusions, as the solver MiraXT [LSB07] showed how including a filter on the clauses that can be received by a solver can significantly change the picture. In this approach, a solver cannot incorporate clauses whose reduct, w.r.t. the current partial assignment, has size greater than 10. Although speedups can be obtained on both satisfiable and unsatisfiable formulae, efficiency for this solver is still below 1.

In 2008 took place the first SAT-race [JLBRS12] with a parallel track for testing multi-threaded parallel SAT solvers. At such competition, it made its appearance the first modern portfolio solver, ManySAT [HJS09b], that is also the winner of that edition. The second position, with 5 instances less solved than the winner solver, was taken by the last divide-and-conquer algorithm, PMinisat [GSH08]. Research over divide-and-conquer solvers stopped that day.

Declared goal behind ManySAT design choice was to take advantage from two of the main weaknesses of divide-and-conquer solvers, namely their sensitivity to parameter tuning and their lack of robustness (i.e., different runs over the same instance are likely to exhibit different run-times). ManySAT differentiates the search by setting each sequential solver with configurations that differs in parameters like restart policies, branching heuristics, random seeds and conflict clause learning. Clause sharing is exploited by allowing each solver to directly communicate with every other solver. ManySAT uses a size-based filter to decide which clauses can be sent; all clauses of size up to 8 are shared. However, communication has a cost and the picture could change when the number of threads increases (the number 8, indeed, was optimised for four threads). That is the reason why in [HJS09a] ManySAT's authors reworked the clause sharing mechanism in such a way that the size of the clauses that can be shared is dynamically decided.

PMinisat, we said, is the last (at the best of author's knowledge) implementation of a (pure) divide-and-conquer algorithm. This solver incorporates all of the main features of previous divide-and-conquer solvers, like dynamic work stealing, search through guiding paths and clause sharing. Moreover, it proposes a series of data-structures that, the authors says, increases the speed of the unit-propagation engine by around 80% [GSH08]. A new idea introduced with PMinisat, instead, is the one of a filter on the clauses that can be received by a thread. Before incorporating a shared clause, each thread evaluates its relevance w.r.t. the guiding-path specific for that thread. If the clause is relevant enough (i.e., it is short w.r.t. the thread current assignment) it gets attached to the learnt clauses for the thread.

Subsequent works on parallel SAT solvers mainly focused on the problem of clause sharing. For example, PeneLoPe [AHJ$^+$12b] offers a fine-grained management of shared clauses and each thread does not handle them as normal learnt clauses. As in *Section* 4.5 we will have a dedicated discussion on achievements in clause sharing, more details can be found there.

4.3 Search Space Partitioning Approaches

A fine-grained division of divide-and-conquer parallel SAT solvers is given in [HJN10]:

- *plain partitioning* [HJN10], where the search space is partitioned into sub-spaces and each sub-space is assigned to a solver.
- *iterative partitioning* [HJN10], where a formula is partitioned iteratively into a tree of sub-problems and each sub-problem is solved in parallel.

Plain partitioning derives from an original formula F a number n of sub-problems $F_, \ldots, F_n$, s.t. :

- $F \equiv (F_1) \vee \ldots \vee (F_n)$
- $F_i \wedge F_j$ is unsatisfiable, for every $i \neq j$

Observe that guiding-path solvers are a special case of plain-partitioning solver, where each partition F_i is a formula of the kind $F \wedge K_i$, and K_i is formula containing a single unit clause. Being a pure divide-and-conquer approach, plain-partitioning suffers from a theoretical slowdown in case the input formula is unsatisfiable [HM12a]. An idea of why this is the case comes from the observation that, in order to prove unsatisfiability of the input formula, unsatisfiability of for each single partition has to be proved first; these partitions are not guaranteed to be easier of the original formula. The original unsatisfiable formula, indeed, could contain a unsatisfiable strict sub-formula: partitioning over a variable not belonging to this sub-formula would result in partitions containing the same unsatisfiable core, that means of equal complexity.

Iterative partitioning does not suffer of this flaw, since the node at the root of the tree (that is, the node working over the original formula) is always running, providing a bound over the overall execution time. Moreover, it can be showed that adding more parallel resources to the procedure, in theory, necessarily leads to an improved run-time [HJN10].

Although portfolio solvers are the current state-of-the-art for what concerns parallel SAT solvers, they have some limit for what concerns scalability [HM12b]. Remarkable in this sense is the work in [AHJ$^+$12a] that manages to achieve good scalability results up to 32 cores.

Iterative partitioning, instead, does not pone a scalability problem: if more resources are available, then more jobs are created in form of partitions of the search space. However, being iterative partitioning only recently proposed, only a study on how to divide the search space [HM12b] and on limited clause sharing [HJN11] has been published. Since the implementation of [GSH08], using guiding-paths, placed not too far behind Manysat in SAT08 competition, we expect that iterative partitioning solvers could, in some years, become competitive against portfolio solvers. In this work we contribute to this goal by improving the clause sharing mechanism presented in [HJN11].

4.4 Iterative Partitioning

How to partition the search space in an iterative partitioning setting is showed in *Figure* 3; moreover, we already defined the requirements for each partition in a plain partitioning approach. Here we formalise better the partitioning mechanism by introducing the notion of *partition function*.

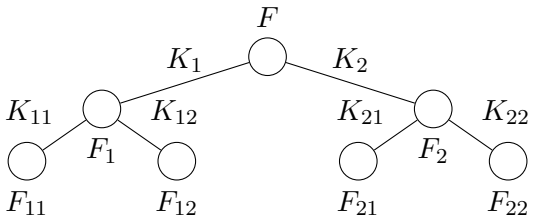

Figure 3: The tree shows how a formula can be partitioned iteratively by using a partitioning function that creates two child formulas.

Definition 4.1. A *partition function* is a function $\phi : \mathcal{F} :\rightarrow 2^{\mathcal{F}}$ that associates to a formula $F \in \mathcal{F}$ an n-ary set of formulae $\{F_1, \ldots, F_n\}$ such that:

- $F \equiv F_1 \vee \ldots \vee F_n$
- for each $1 \leq j < i \leq n$, $F_i \wedge F_n = \bot$

Looking at the SAT problem for a formula F as a search problem, the search space can be represented as the set of all assignments over the propositional variables $atom(F)$. A partition function divides this search space into a certain number of partitions. Without loss of generality, from now on we assume that every partition $F_i \in \phi(F)$ is of the form $F \wedge K_i$, for some *formula constraint* K_i. The idea of iterative partitioning is that the application of the partition function can be iterated over child formulae. This process result in the creation of a so called *partition tree*:

Definition 4.2. A *partition tree* for a formula F w.r.t. a partition function ϕ is a tree $\mathcal{T}$ rooted in F such that, for every node $F' \in \mathcal{T}$, the set of its direct successors is $\phi(F')$.

For our purposes, it is convenient to mark each node of a partition tree with a *position*; the root node has position ϵ, whereas the node at position pi is the i-th successor of the node at position p. Without loss of generality, for every partition tree $\mathcal{T} := (\mathcal{N}, \mathcal{R})$ we assume that each i-th successor of a node $F' \in \mathcal{N}$ is a positive number in base $|\phi(F')|$. With $pos(\mathcal{T})$ we denote the *set of positions* in the partition tree $\mathcal{T}$. Over such a set it is possible to define a *prefix* relation.

Definition 4.3. Consider a set $pos(\mathcal{T})$ of positions in a partition tree $\mathcal{T}$. Then we define the *prefix* relation $\leq \subseteq pos(\mathcal{T}) \times pos(\mathcal{T})$ as:

$$p \leq q \text{ iff } \exists q' \in pos(\mathcal{T}) \text{ s.t. } q := pq'$$

From now on, the notation F^p will indicate the formula at position p of a partition tree rooted in F. An useful notion is the one of *length* of a position.

Definition 4.4. Consider a partition tree $\mathcal{T} := (\mathcal{N}, \mathcal{R})$ for a formula F w.r.t. a certain partition function ϕ. Let $p \in pos(\mathcal{T})$ be a position in the partition tree. Then

the *length* $|p|$ of p in this tree is defined as:

$$|p| := \begin{cases} 0 & \text{, if } p = \epsilon \\ |q| + 1 & \text{, if } p = qi \text{ and } (F^q, F^{qi}) \in \mathcal{R} \end{cases}$$

Observe that, for each position $p \in pos(\mathcal{T})$ valid position in a tree $\mathcal{T}$, it holds $F^p = F \cup K^{i_1} \cup K^{i_1 i_2} \cup \ldots \cup K^{i_1 \ldots i_n}$, if $p := i_1 \ldots i_n$ and each $i_j \in \{1, \ldots, |\phi(F^{i_1 \ldots i_{j-1}})|\}$. Additionally, since a partition tree is created upon a partition function, it directly follows from *Definition* 4.1 that, for every $p \in pos(\mathcal{T})$ and $i, j \in \{1, \ldots, |\phi(F^p)|\}$ with $i \neq j$:

- $F^p \equiv \bigvee_i F^{pi}$ and
- $F^{pi} \wedge F^{pj} \equiv \bot$

Sharing learnt clauses among solvers that solve child formulae has been considered in [HJN11]. There, Hyvärinen et. al. introduce an expensive mechanism called *assumption (learnt) clause tagging* and a fast approximation method called *flag-based (learnt) clause tagging*. Next section is devoted to clause sharing in iterative-partitioning, and the aforementioned methods are talked in more detail. Our contribution is a rework of the second method that incorporates strengths of the two methods, outperforming both.

4.5 Clause Sharing so Far

Due to the fact that it is an easy extension of every CDCL based parallel SAT solver, it is no surprise that many approaches exploit the clause sharing technique.

As we said, PaSAT [SBK01] is the first shared-memory solver that implements clause sharing. In detail, PaSAT uses a *Global Lemma Store*, held in shared-memory, where all clauses that can be shared are kept. The criterion to send a clause to the global lemma store is a fixed size one. In regular intervals the prover tasks check if there are new clauses in the global store, and if this is the case, they integrate them into their clause sets.

While claiming that it is not clear how much a solver can profit from sharing clauses, authors report occasional speed-ups over SAT formulae as well as cases in which the solver becomes slower. Efficiency over unsat instances is close to 1, regardless clause-sharing activation.

A different approach is taken in GridSAT [CW03], where instead of shared memory the solver makes use of a cluster of network connected machines. Rather than keeping shared clauses in a global lemma store, sharing in GridSAT takes place at the act of jobs creation, as the father job propagates all the clauses learnt so far to its children jobs. Moreover, short clauses are sent to other clients each time they are learnt. An obvious drawback of this approach is the fact that clauses of a client are discarded once it finishes to analyse an unsatisfiable sub-problem. Nevertheless, occasional speed-ups for both satisfiable and unsatisfiable instances are observed, as well as occasional slow-downs. Like PaSAT, a solver can merge shared clauses with its own set of clauses only at regular intervals (i.e., only at restarts).

Authors of PMSat [GFS08] took inspiration from GridSAT, but it proposes a substantially different sharing mechanism that looks more like the one in PaSAT. Each time a client in PMSat finishes its job by proving unsatisfiability, a subset of its smallest and "more active" clauses is sent back to the master. Observe that this overcomes the problem of GridSAT by allowing shared clauses to "outlive" the client where they were learnt. The master is in charge of the propagation of shared clauses to each client in the grid. Though authors of PMSat points out the fact that a fair comparison between their approach and previous ones is impossible, they report a higher percentage of super-linear speedups when compared with the results presented by previous solvers.

All the approaches discussed so far try to limit the number of clauses that can be shared, by means of a filter on the clause size. We already saw how in multithread solver ySAT [FDH05], instead, every learnt clause is physically shared. This is done by maintaining a list of generated learnt clauses that is accessible from all threads. When a thread generates a learnt clause, it puts this clause onto the list. The other threads frequently check the existence of new clauses on this list. Once the clause has been acquired by every thread, the clause is removed from the list. Authors claim that the overhead due to access of all the threads to a single shared clauses store is "insignificant compared to the overall performance". Even though experiments were taken with a few cores, author of this thesis does not agree with such a claim at all, nor their results do. Another quite hazardous claim is in the conclusion of their work, as they claim that there seems to be no practical advantage in attempting to optimise a CDCL algorithm by letting it execute concurrently on a multiprocessor workstation.

For the first work on clause sharing going beyond a mere fixed-size filter one has to wait for the first portfolio solver, and especially [HJS09a] is an important milestone in the field of clause sharing. In this work Hamadi et al. point out three main flaws of every fixed-size clause sharing technique. The first is related to the fact that it is difficult to predict the "ideal" size; a too small size might completely inhibit the cooperation, while an overestimated one may induce a very large cooperation overhead. The second flaw comes from the observation that, in modern CDCL solvers, the size of learnt clauses tends to increase as the computation goes on. In a fixed size clause sharing setting, then, a sudden halt of cooperation might occur eventually (*saturation problem*). The third flaw is related to the fact that it is hard to predict if a clause will be useful for the receiving solver, since different restart policies and activity scores could lead two different search processes towards completely different search spaces and make the cooperation pointless (*relevance problem*).

Hamadi observes that the first two flaws can be solved by maintaining the *throughput* for each receiving solver constant over time. Throughput is defined as the number of foreign clauses received in each time interval, where the time interval is typically defined in terms of a fixed number of conflicts. Each solver keeps an upper bound and a lower bound on the sizes of the clauses that can be received. If the current throughput is higher than the ideal value, then a uniform decrease of the limits is performed in order to decrease the cooperation with other solvers and the communication cost. Likewise, if the current throughput is less than the ideal value then limits are uniformly increased in order to enhance cooperation with other solvers.

The third flaw, instead, can be solved by introducing a *quality measure* over the clauses that are received. Instead of maintaining a constant throughput on incoming clauses, each solver tries to maintain a constant throughput of a fixed quality instead. The quality measure used in [HJS09a] is obtained by comparing the activities associated to each literal in the clauses that are received within the current time interval to the maximum activity associated to a literal from the receiving solver.

Experimental results shows that these techniques result in an improvement of the solver performance, as they manage to solve 6 instances more than plain ManySAT over SAT Race 2008 parallel track industrial instances, and 7 more on crafted instances.

A more recent paper from the same authors [LHJS12] instead is concerned with controlling the cooperation network, that is, selecting the solvers referred to as *emitters* that are allowed to send clauses to any receiver solver. In a classical portfolio setting with clause-sharing, indeed, each solver can typically communicate with every other solver that is working in parallel. This means that, considering the set of solvers as a graph where each node is a solver and an edge between two nodes corresponds to a connection between them, the topology of the network is the one of a clique. Lazaar et al. point out how this kind of topology is quite disadvantageous as communication costs become a critical issue. For this reason, they propose an approach that dynamically controls the topology of the network, based on *Multi-Armed Bandit* formalisation for emitter selection. In practice, each receiver solvers associates to each emitter a certain probability giving more weight to those solvers that are emitting "better clauses" (w.r.t. to the search of the receiver). Each time a solver tries to incorporate shared clauses, emitters will be chosen according to the probabilities associated to them. Experimental results on SAT Challenge 2012 industrial instances show that this modification can significantly improve the performance of a parallel portfolio solver.

Another question of great relevance is related to the life of a shared clause once it is incorporated among the clauses of the receiving solver. Every sequential CDCL solver, indeed, usually performs some kind of cleaning strategy in order to prevent the database of learnt clauses from growing to an unmanageable size. Since these cleaning strategies are designed having a sequential scenario in mind, it becomes difficult for a thread to manage clauses that have been generated from other solvers. A recent work from Audemard et al. [AHJ$^+$12a] makes a study on the whole life-cycle of a shared clause once it is incorporated in the learnt database of a receiver solver. They point out how, with the cleaning strategy of MiniSat 2.2, most of these clauses are discarded without contributing to the search process at all.

In order to overcome this problem, they introduce a measure called *psm* to give to each received clause a score. A clause that is likely to participate to the search, e.g. a clause that is near to become unit, is given a higher score than a clause that most likely will not participate to the search. Instead of the usual cleaning strategy, clauses acquired through sharing mechanism are handled in a particular way. First, they extend MiniSat in order to give it the ability to *freeze* [ALMS11] clauses. A frozen clause is a clause that cannot participate in unit propagation steps of the solver. This means that the data structure usually kept for unit propagation and related to that clause can be discarded, thus saving resources. Frozen clauses

can be reactivated again in future, if their psm score becomes promising again. In PeneLoPe [AHJ+12b], the winner of the last SAT competition, a clause is definitely deleted if it stays frozen for more than 7 restarts. The final results of [AHJ+12a] shows very interesting insights, as they manage to obtain a portfolio procedure able to scale up to 32 cores. Unfortunately, every SAT competition organised so far did not go beyond 8 cores.

Observe that in this digression over the evolution of clause sharing during years we purposely left aside iterative partitioning SAT solvers. The reason for this is that clause sharing in an iterative partitioning setting is problematic, as it might harm the soundness of the whole parallel procedure. In order to treat the subject thoroughly, the rest of *Section* 4 is devoted to this topic.

4.6 Clause Sharing in Iterative Partitioning Solvers

In approaches based on guiding-paths, constraints are always in the form of unit clauses. Internally to a SAT solver, unary clauses are usually not treated as clauses but simply added to the current partial assignment as literals of level zero. This means that, unless clauses are *simplified* w.r.t. to the level zero literals (as it happens in MiniSat), clauses learnt in any sequential solver can be obtained as a resolution derivation over clauses from the original formula. This means that every learnt clause is a semantic consequence of the original problem, that implies that every learnt clause is a semantic consequence of every partition and can be sent to any other solver running in parallel.

Clearly, a solver cannot receive a clause that could make its local sequential procedure unsound. For example, clauses specific for a certain partition (e.g., clauses belonging to a partitioning constraint) cannot be sent to a different partition of the search space, because, assuming that the formula being solved from the receiving solver is satisfiable, there could be no model that satisfies both the sent clause and the formula of the receiving solver. In other words, every receiver working on a certain formula cannot receive clauses that are not a semantic consequence of that formula.

It is arguable whether it is a good idea to forbid simplification steps for the only sake of clause sharing. However, even by doing so things would not work in our setting, where constraints are generic formulae and do not necessarily have to be unary clauses. This allows for clauses belonging to a partitioning constraint to participate in the conflict analysis to produce new learnt clauses, leading so to learnts that not necessarily are a semantic consequence of the original problem. An example of this problem is given in *Example* 4.5.

Example 4.5. Consider the partition tree in *Figure* 4. Observe that, in order to better represent the simplifications that MiniSat performs with the clauses of the original formula and literals of level zero, we assumed that nodes F^1 and F^2 are respectively the reducts $F|_{(\neg 1)}$ and $F|_{(1)}$ of the formula F at the root of the partition tree. Of course, the clause $\{\neg 1\}$ cannot be sent to the solver working over F^2, since $F^2 \not\models \{\neg 1\}$. This is not the only clause that F^1 cannot send to F^2. Indeed, consider the formula F^1 in the partition tree and the following local run of a CDCL solver

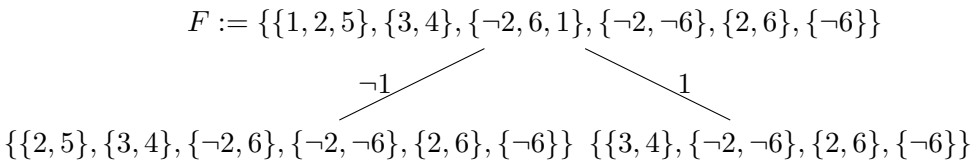

Figure 4: Partition tree for F

working over F^1:

$$F^1 :: (\,) \rightsquigarrow_{dec} F^1 :: (\neg\dot{5}) \rightsquigarrow_{unit} F^1 :: (\neg\dot{5}, 2) \rightsquigarrow_{unit} F^1 :: (\neg\dot{5}, 2, 6)$$

Observe, this run leads to a conflict after the decision $\neg\dot{5}$ and unit propagations 2 and 6 so that the clause $\{\neg 2\}$ is learnt. Since $F^2 \not\models \{\neg 2\}$, this clause cannot be added to the clauses of F^2. Moreover, observe that adding it would affect the soundness of the sequential procedure solving F^2, since F^2 is satisfiable and $F^2 \models \{2\}$. For the same reason, $\{\neg 2\}$ cannot be sent to the solver working over the original formula F.

4.6.1 Assumption-based Clause Tagging

Example 4.5 makes it clear that, in order to guarantee soundness, one has to give the solver the ability to "remember" the fact that a particular clause *depends* on a certain partitioning constraint. The idea of *assumption-based clause tagging* [HJN11] is to add to each clause in a partitioning constraint K_i a new negated propositional variable $\neg a_{K_i}$. Moreover, the unary clause $\{a_{K_i}\}$ is added to K_i as well. First, observe that a SAT solver working over the partition defined by K_i propagates the literal a_{K_i} before anything (or better, before performing any decision). Once this is done, the local search on that solver goes on normally. Notice that these "tagging literals" will be carried on when learning a new clause. The next lemma explains why these tags are important, and gives a hint on how they could be used in order to perform "safe" clause sharing:

Lemma 4.6. *Let $\mathcal{T} := (\mathcal{N}, \mathcal{R})$ be a partition tree rooted in a formula F. Consider a node $F^p \in \mathcal{T}$, and let C be a clause containing a number of negated tag literals $\neg a_{K^{p_1}}, \ldots, \neg a_{K^{p_n}}$. Then $F \wedge K^{p_1} \wedge \ldots \wedge K^{p_n} \models C$.*

Proof. In case C is a partitioning constraint, the conclusion follows directly from how tagging is defined. If C is a clause learnt in some step by the local solver, then it has been obtained as a resolution from clauses in $F \cup K^{p_1} \cup \ldots \cup K^{p_n}$, since tag literals are always newly introduced literals. Hence the result. □

Now, observe that constraints in *Lemma* 4.6 are s.t. $p_1, \ldots, p_n$ are prefixes of p (from construction of the partition tree $\mathcal{T}$). Moreover, among these prefixes there is one of maximal length p_j s.t. p_i is a prefix of p_j, for every $i \neq j$. Thus, clause C is a semantic consequence of every successor node of F^{p_j}, that is, this clause can be shared among all the solvers working over this partition.

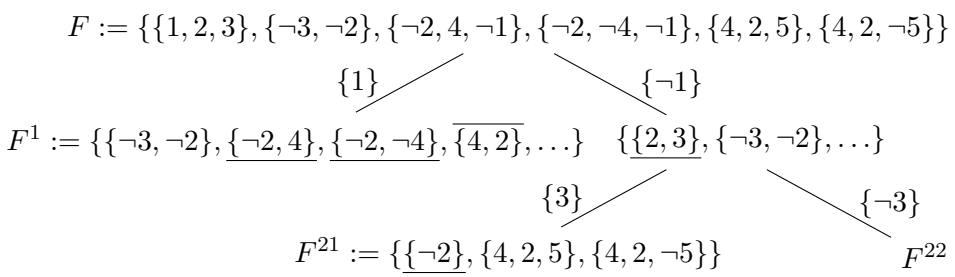

Figure 5: Partition tree over F with clause-tagging. Unsafe clauses are underlined. The overlined clause $\overline{\{4,2\}} \in F^1$ is a shared clause that has been incorporated from F^{21}.

Although assumption-based tagging is indeed a quite elegant approach, it has been observed in [HJN11] that it leads to longer clauses (due to the tagging literals) that significantly worsen the performance of the overall procedure, in terms of speed and memory consumption. In fact, assumption-based clause tagging in less efficient than an approach where sharing is not used at all. In order to overcome this problem, Hyvärinen proposes a fast approximation of assumption-based tagging that does not make use of assumption literals and thus does not introduce significant overhead. This last approach, that represents the current state of the art concerning clause sharing in iterative partitioning setting, is called *flag-based* clause tagging.

4.6.2 Flag-based Clause Tagging

In *flag-based clause tagging*, clauses are associated a flag that determines whether they can be globally shared, or not. Clauses can be flagged as "safe", that means they do not depend on partitioning constraints, or as "unsafe" otherwise. This approach has the advantage that no new literals have to be included in clauses, resulting in a better memory usage than assumption-based. However, less clauses can be shared than assumption-based, since a boolean flag cannot identify the partitioning constraints the learnt clause depends on. Flags are assigned in the obvious way, that is a clause is tagged as "unsafe" only if it is part of a partitioning constraint or it has been obtained with a resolution derivation involving clauses belonging to partitioning constraints. One can formally prove that every safe clause is a semantic consequence of the formula at the root of the partition tree, implying that each safe clause can be received from any solver in the partition tree. Next example shows how clauses received by a node with flag-based sharing can effectively speed-up the computation on that node.

Example 4.7 (Flag-based Sharing)**.** *Figure* 5 shows an example of a partition tree in which unsafe clauses are underlined. Consider the following CDCL execution for F^{21}, which yields the conflict clause $\{4,2,\neg 5\}$:

$$F^{21} :: () \rightsquigarrow_{unit} F^{21} :: (\neg 2) \rightsquigarrow_{dec} F^{21} :: (\neg 2, \neg\dot{4}) \rightsquigarrow_{unit} F^{21} :: (\neg 2, \neg\dot{4}, 5)$$

At this point, by performing a conflict analysis starting from the conflict clause

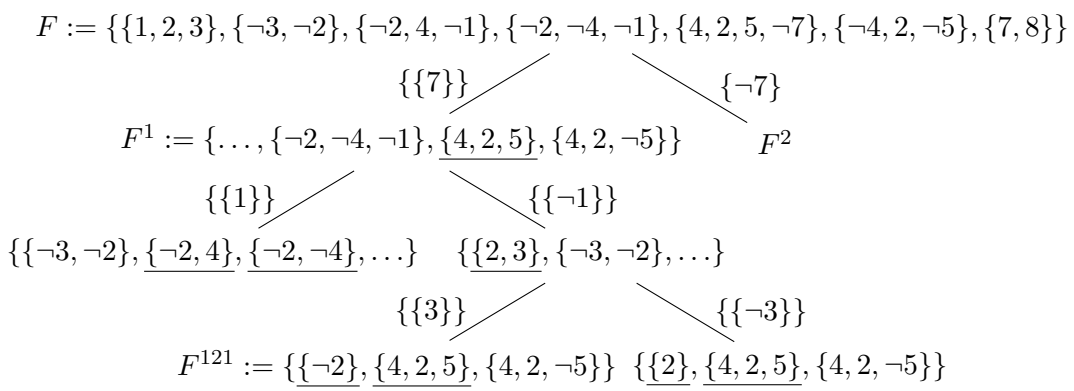

Figure 6: Clause $\{4,2\}$, learnt by the local solver working on the node solving formula F^{121}, is not safe anymore, because it depends on the partition constraint $\{\{7\}\}$.

$\{4,2,\neg 5\}$ one can derive the learnt clause $D := \{4,2\} = \{4,2,5\} \otimes \{4,2,\neg 5\}$. Since only safe clauses have been used during resolution, D is a safe clause and thus it can be shared among every node in the partition tree. In *Figure* 5 D is received from the solver working over the formula F^1. Observe that clause $\{4,2\}$ speeds-up the computation on node F^1. Indeed, consider the following sequential execution over node F^1 after incorporating the shared clause $\{4,2\}$:

$$F^1 :: (\ ) \rightsquigarrow_{dec} F^1 :: (\neg\dot{4}) \rightsquigarrow_{unit}$$

$$F^1 :: (\neg\dot{4},\neg 2) \rightsquigarrow_{back} F^1 :: (\ ) \rightsquigarrow_{learn} F^1, \{4\} :: (\ )$$

After the decision $\neg\dot{4}$, the local solver can immediately use the shared clause $\{4,2\}$ to derive the learnt clause $\{4\}$. Performing the same decisions and propagating without using the safe shared clause, instead, would lead to the learnt clause $\{4,2\}$. Hence, flag-based clause sharing can effectively speed-up the local computation of some node in the tree.

We said that flag-based sharing is an incomplete approximation of what could be shared "safely". An example of this fact be seen in *Figure* 6, where we slightly changed the shape of the partition tree. Consider such a partition tree and, like we did for *Example* 4.7, assume that the clause $D = \{4,2\}$ is learnt while working on formula F^{121}. Since the resolution $\underline{\{4,2,5\}} \otimes \{4,2,\neg 5\}$ involves an unsafe clause, D is also tagged as unsafe and thus it is not shared at all. However, from *Example* 4.7 we know that this clause can be "safely" shared among all the nodes F^{1p}, for all positions p of the tree rooted in F. This example illustrates that flag-based tagging is a limited approximation of clause sharing. The problem comes from the fact that flag-based sharing keeps the information about whether a clause depends on some partitioning constraints, without specifying what these constraints are.

Despite these weaknesses, evaluation in [HJN11] shows how flag-based clause tag-

ging is a significant advance from assumption-based, and it is also superior to an approach not exploiting sharing at all. The idea proposed here is to extend the tagging in a way such that it becomes possible to identify what are the partitioning constraints that contributed to the creation of a certain unsafe clause, as it happens for the assumption-based approach, while keeping an overhead similar to the flag-based approach. The new technique is called *position-based clause tagging*.

4.7 Position-based Clause Tagging: a New Approach to Clause Sharing

Flag-based sharing is designed in a way that a clause can be shared only if this clause is a semantic consequence of the original formula. In other words, unsafe clauses that are semantic consequences of formulae belonging to some strict sub-tree of the partition tree are not shared at all. If the tag encodes the sub-tree where a clause is "*safe*", this clause can at least be shared in this sub-tree. The key idea of position-based tagging is to associate each clause a position in the partition tree. If C is a clause, and p a position in the partition tree, C^p denotes that the clause C is tagged with the position p. Given a partition tree $\mathcal{T}$ for a formula $\mathcal{F}$, clauses belonging to $\mathcal{F}$ have to be tagged with the empty position ϵ. Clauses in a partition constraint K^p are tagged with the position p. A learnt clause D^q obtained from a resolution derivation $(R_1^{p_1}, \ldots, R_n^{p_n})$ is tagged with the longest position q among the positions of the clauses that are used for resolution, i.e. $q = \arg\max_{p_i} |p_i|$, where $1 \leq i \leq n$. Observe that the same clause can be derived in different nodes of the partition tree and thus be given different positions. In order to permit a sequential solver to *receive* a clause from another node in the same partition tree, the $DPLL$ reduction system presented in *table* 1 needs to be extended.

First, observe that the tagging is defined in terms of resolution steps. Thus, the generic *learn* rule of the $\mathcal{CDCL}$ abstract reduction system needs to be replaced with one that enforces learnt clauses to be obtained through resolution steps, only. Observe that this is not a limitation w.r.t. real word implementations of CDCL solvers, since they all use resolution to derive learnt clauses (see *Section* 3.2). The second step is to give each sequential solver the ability to receive clauses from other computation units. This is done through the *receive* rule:

Definition 4.8. Let G be a formula, J a partial interpretation and F^{pq} the node at position pq of a partition tree rooted in F. Consider a clause C^p. Then

$$G :: J \quad \rightsquigarrow_{rec} \quad G, C^p :: J \text{ iff } F^{pq} :: () \overset{*}{\rightsquigarrow} G :: J$$

Note that the position p of the clause C^p is a prefix of the position pq of the formula F^{pq}. The correctness of this receive rule is obtained by showing that the formula F^{pq} entails any clause C^p, which we formally state as Corollary 4.13 below. In order to prove it, we make use of an auxiliary definition:

Definition 4.9. Let F^q be a node in a partition tree rooted in F. Consider a sequential chain $F^q :: () \overset{*}{\rightsquigarrow} G :: J$ s.t. $C^p \in G$. Consider a clause R^s. Then $C^p >_{res} R^s$ iff C^p is a learnt clause and R^s is one of the resolvents used to derive C^p.

It is not hard to see that the transitive closure $>_{res}^{+}$ of $>_{res}$ is a well-founded strict partial order, since each learnt clause is the result of a finite resolution derivation and each partition tree is finite. Thus, the well-founded induction principle [BN98] is valid on $>_{res}^{+}$.

Lemma 4.10. *Consider a node F^q, and a sequential chain $F^q :: () \overset{*}{\rightsquigarrow} G :: J$ such that $C^p \in G$. Then p is a prefix of q.*

Proof. By well-founded induction w.r.t. $>_{res}^{+}$. (IB) If C^p is not learnt, then it must be $C^p \in F^q$ and thus thesis follows from construction. (IH+IS) Assume C^p has been obtained in some node with a resolution derivation $(R_1^{r_1}, \dots R_j^p, \dots, R_n^{r_n})$, and that the theorem holds for each of these resolvents. If C^p is a received clause, then it must be $p \leq q$ by definition of the receive rule. If C^p has been learnt in F^q, then the lemma hypotheses hold for R_j^p as well, and thus from (IH) p is a prefix of q. □

Lemma 4.11. *If C^p is a learnt clause that has been obtained from a resolution derivation $(R_1^{p_1}, \dots, R_n^{p_n})$, then p_i is a prefix of p, for every $1 \leq i \leq n$.*

Proof. This is a consequence of Lemma 4.10 □

Theorem 4.12. *Let F^p be a node of a partition tree $\mathcal{T}$ rooted in F. Let C^p be a clause belonging to some node of $\mathcal{T}$. Then it holds $F^p \models C^p$.*

Proof. By well-founded induction w.r.t. $>_{res}^{+}$. (IB) If C^p is not learnt, then $C^p \in F^p$, and thus $F^p \models C^p$. (IH + IS) Assume C^p is obtained as a resolution from resolvents $(R_1^{q_1}, \dots, R_n^{q_n})$, and that the theorem holds for each of these resolvents. From Lemma 4.11, we have that $q_1, \dots, q_n$ are prefixes of p. This, together with the definition of partition tree, leads to:

$$F^{q_i} \subseteq F^p, \text{for each } 1 \leq i \leq n$$

Thus $F^p \models F^{q_i}$, for $1 \leq i \leq n$. From (IH) and transitivity we derive that F^p models every resolvent of C^p, concluding that $F^p \models C^p$. □

Corollary 4.13. *Let F^{pq} be a node in the partition tree $\mathcal{T}$. Let C^p be a clause belonging some node of $\mathcal{T}$. Then it holds $F^{pq} \models C^p$.*

Proof. It follows directly from Theorem 4.12 and equality $F^{pq} = F^p \cup K^{pq}$. □

Observe how *Corollary* 4.13 provides a way to show soundness for an iterative approach exploiting position-based clause sharing, as it guarantees that each clause received by a node is a semantic consequence of that node. Moreover, the theorem shows how each safe clause in flag-based approach is a semantic consequence of each receiver node, since unsafe clauses in our setting would simply correspond to clauses tagged with a non-empty position. Please notice that this is the first formal proof of correctness for flag-based clause tagging, as work in [HJN11] presents the approach without providing a proof of its correctness.

Example 4.14. Now reconsider the example in *Figure* 6, which is an extension of *Figure* 5. Flag-based clause tagging was not able to share the learnt clause $\{4, 2\}$ anymore, because $\{4, 2\}$ unsafe. The new sharing rule position-based tagging can share this clause again as in the situation of *Figure* 5: all solvers working on formulae F^{1p}, where p is a position such that $1p \in pos(\mathcal{T})$, can receive clause $\{4, 2\}$.

5 Empirical Evaluation

5.1 Underlying Iterative Partitioning Solver

At the time of writing, and at the best of author's knowledge, the only multi-threaded parallel SAT solver exploiting the iterative partitioning strategy is *splitter* [BBD+12]. Splitter uses MiniSat 2.2 [ES03] as underlying sequential solver, where each of these solvers is submitted *jobs* (alias, partition tree nodes) from a master thread. Each time more jobs are needed, a node in the partition tree is chosen to be split and given to a *splitter* thread which applies a certain partitioning function on it. The master is responsible for generating a sufficient number of jobs such that the parallel resources, e.g. the available cores in a multi-core architecture, are saturated.

All of the experiments here presented make use of a fixed basic configuration chosen among the ones which seemed to perform best over the SAT Challenge 2012[3] set of instances. This configuration differs from the one that has been submitted, together with splitter, to the last SAT Challenge (a description of the configuration can be found in [BBD+12]), since that configuration was so bad that relegated splitter in the last position (by far) of the whole competition.

The configuration we adopt makes use of a *conflict-based* limit in order to stop those nodes that seem to be assigned a too difficult partition. In the tests, this limit is set to 512000 conflicts, i.e. every 512000 clauses learnt in a node the search over that node is stopped and the node is split.

The partitioning function used for the tests is the default one of splitter: *VSIDS Scattering*.

5.1.1 VSIDS Scattering

VSIDS scattering is a method for partitioning nodes in a partition tree proposed by Hyvärinen in [HJN06]. Some auxiliary notions are needed in order to introduce the scattering method, namely the notions of *cubes* [HKWB12] and *scattering partition function*.

Definition 5.1. A *cube* is a formula $Q := \{C_1, \ldots, C_n\}$ such that $|C_i| = 1$, for each $1 \leq i \leq n$.

Please observe that the negation of a cube $Q := \{\{L_1\}, \ldots, \{L_n\}\}$ is the clause $\{\neg L_1, \ldots, \neg L_n\}$.

Definition 5.2. An n-ary *scattering function* is a function that, by generating $n-1$ cubes $Q_1, \ldots, Q_{n-1}$, partitions a formula F_0 into n partitions $F_1, \ldots, F_n$ defined as:

$$\begin{cases} F_1 & := F_0 \wedge Q_1 \\ F_{m+1} & := F_0 \wedge \bigwedge_{i=1}^{m} \neg Q_i \wedge Q_{m+1}, \text{ for } 1 \leq m < n-1 \\ F_n & := F_0 \wedge \bigwedge_{i=1}^{n-1} \neg Q_i \end{cases}$$

[3] http://baldur.iti.kit.edu/SAT-Challenge-2012/

It follows directly from definition that the hardness of the scattering procedure relies in the cubes generation phase. The aim is to produce partitions that have similar estimated computation costs in order to avoid trivially unsatisfiable nodes and thus prevent the partition tree from growing too much. Nodes generation phase, indeed, is costly and generating too easy nodes would not amortise the overhead introduced for their creation (see *ping-pong* effect [JLU01]).

For generating the cube K_i ($i < n$), the partition F_{i-1} is given in input to a splitter thread. The splitter thread will launch a MiniSat instantiation over that partition, and let it run until a node $F'_{i-i} :: J$ is met such that the number of decision literals in J is i. Let $\{d_1, \ldots, d_i\}$ be the set of such decision literals. Then the cube K_i is the formula $\{\{d_1\}, \ldots, \{d_i\}\}$.

5.2 Clause Sharing Implementation

5.2.1 From Position-Based Clause Tagging to Level-Based Clause Tagging

The key idea of position-based clause tagging is to tag each clause C with a position in order to identify that (unique) node in the partition tree that minimally contains all of the partitioning constraints used for the generation of C. Thus, as long as this node can be uniquely determined, it is not necessary to use positions to identify it. An alternative tagging that has been used for the experiments, and that is equivalent to position-based tagging, exploits the idea of tagging each clause with an integer representing a *level* in the partition tree.

Definition 5.3. Consider a node F^p in a partition tree $\mathcal{T}$ for a formula F. Then the *level* of F^p in $\mathcal{T}$ is defined as $|p|$.

The semantics we associate to a level tag is the relation "ancestor of level". Observe that, given a node of position p in a partition tree $\mathcal{T}$, its (unique) ancestor of level n can be uniquely determined, as it is the node at position q such that $p = qr$ and $|q| = n$.

Tagging with levels is defined in the obvious way, that is clauses at root node are tagged with level 0; clauses belonging to a partitioning constraint K^p are tagged with $|p|$, and each learnt clause obtained from a resolution derivation $(R^{n_1}, \ldots, R^{n_m})$ is tagged with $max_{n_i}|n_i|$, where $1 \leq i \leq m$.

Clearly, the approach guarantees soundness if sharing is performed complying to *Corollary* 4.13. This means that a clause C^n learnt at a node F^p is a semantic consequence of that (unique) ancestor node F^q such that q is a prefix of p with $|q| = n$.

5.2.2 Pools of Shared Clauses

Each node in the partition tree is associated a *pool* containing shared clauses tagged with the position of that node, and only those. Given a node F^p we refer to its pool as the *pool of position p*. Being pools entities separated from solvers threads, shared clauses can outlive the solver where they have been produced.

From now on, we refer as *pull* to the action that a solver performs in order to get a clause from a pool of shared clauses; likewise, we call *push* the action that a solver performs in order to write a new clause in a pool of shared clauses.

For an easier implementation solvers are allowed to pull shared clauses from a pool only at restarts. According to *Corollary* 4.13, each solver working over a node F^p can only pull clauses from pools of position $q \leq p$, that is, from all of the ancestors of the node it is solving. Since it has been observed that the height of the tree tends not to grow too much (see *Table* 4), in the evaluation here described solvers are allowed, at every restart, to pull clauses from each pool they can access.

Pools are implemented as *ring-buffers* with a fixed maximum size, common to every pool. This means that a pool is allowed to grow until a certain maximum value is reached: from this point onwards, new incoming clauses replace the oldest clauses in the pool.

Since the majority of the clauses is sent to the (unique) pool of level zero, also because in general the majority of learnt clauses is safe, pools downwards in the tree might be scarcely populated. However, observe that allowing a pool at position q to contain clauses of position $p \leq q$ would not violate the hypotheses of *Corollary* 4.13. Thus, one could obtain a more homogeneous "pools filling" by artificially worsening (alias, increasing) the levels of shared clauses and, consequently, sending them downer in the tree (we called this technique *dynamic level*).

Tests tell that *dynamic level* helps in case the pools have a relatively small maximum size, but not in the case where pools are allowed to grow up to a considerable size before starting behaving as a ring-buffers. When pools are small, indeed, clauses are deleted often and for this reason they are prevented from contributing much to the search. In this scenario dynamic level option can mitigate the problem by optimising the way pools are filled.

5.2.3 Handling of Unit Clauses

Each clause obtained through a resolution derivation is tagged as prescribed from position-based tagging (in its level-based variant). A special mention, however, is deserved by the handling of unit clauses. By default MiniSat does consider unit clauses as level zero literals, and thus it does not keep a clause structure for each unit clause. It follows that level zero literals in MiniSat can participate to resolution steps as every normal unit clause would do. Consequently, each level zero literal needs to be tagged as if it was a normal clause. With this respect, unit propagation steps are critical since unit propagation at level zero can be seen as a resolution derivation. This duality between unit propagation and resolution is exposed by *Example* 5.4.

Example 5.4. Consider a unit propagation step $F :: P \rightsquigarrow_{unit} F :: P, L$, where P is a sequence of level zero literals, and consider the clause C in F such that $C|_p = \{L\}$. Observe that the unit clause $\{L\}$ can alternatively be obtained as a resolution derivation $(R_1 := C \otimes \{\overline{L_1}\}^{p_1}, \ldots, R_n := R_{n-1} \otimes \{\overline{L_n}\}^{p_n})$ where each $L_i \in C \backslash C|_p$ $(1 \leq i \leq n)$. Thus, according to *Corollary* 4.13, clause $\{L\}$ has to be tagged with $\arg\max_{p_i} |p_i|$, where $1 \leq i \leq n$.

Since unit-propagation is a very crucial part of the sequential CDCL procedure, this kind of analysis is not very convenient as it introduces overhead over the unit-propagation engine. Nevertheless, empirical results show that this does not prevent configurations with clause sharing from being faster than configurations were clause sharing was disabled (*see Sections* 5.4 and 5.5).

5.2.4 Pulling Clauses

After pulling a shared clause from a pool, the solver copies it into its local database of learnt clauses. In order to prevent the clause from being deleted too early without contributing to the local search, the activity for that clause is increased once (MiniSat uses an activity based clause removal, see *Section* 3.3.4 or [ES03]). As discussed in *Section* 4.5, work in [AHJ+12a] points out how this is not an optimal strategy, and that substantially better results could be obtained by introducing a dedicated mechanism for handling shared clauses instead of considering them as normal learnt clauses. This might be an interesting direction for future work.

When receiving a clause, MiniSat by default deletes all the false literals of level zero in that clause. As discussed in *Section* 5.2.3, when this action is performed the level of the clause being received might get worsened. Author tried out to forbid this simplification on shared clauses whenever it would worsen the level of the clause being received. However, no improvements were observed, and the solver seemed to perform worse. One reason for this might be that forbidding simplifications led to longer clauses, worsening the overall performance for the receiver sequential solver. For these reasons in the experiments here presented the default simplification of MiniSat is turned on, and levels of received clauses are allowed to be worsened.

5.2.5 Pushing Clauses

Each solver shares its learnt clauses every a fixed number of conflicts, called *sharing-delay*. Observe that if sharing-delay is set to 1, then each time a clause C^p is learnt that clause is immediately pushed into the pool of position p in the tree. This might introduce a communication overhead due to several (short) lock requests on the pools, as we will see in *Section* 5.5.1.

By default pools are kept duplicates-free: a solver is allowed to push a clause in a pool only if that clause is not already present in the pool. Observe that duplicates check efficiency is strictly correlated to the size of the pools of shared clauses (since the clause to be pushed has to be checked against each other clause in the pool). *Tables* 3 and 8 show how the occurrences of duplicates vary significantly from configuration to configuration. However, in the majority of our experiments author observed that the number of duplicates tends to stay reasonably low, and thus duplicates check might be disabled without negatively affecting the overall performance. A confirmation of this comes from the fact that, by running experiments, we did not observe any significant difference between having the duplicates check feature on or off (in case the pool is small, as in *Section* 5.4). The results here presented are all obtained from configurations where duplicate check was enabled.

5.3 Instances Selection

Experiments have been performed on two different instances sets. The first one, counting 600 instances, is the whole set of industrial instances from SAT Challenge 2012 [JLBRS12]. From now on, we will refer to this set simply as SAT12. The second one is the set of industrial instances from SAT Competition 2009, consisting of 292 instances and that corresponds to the same instance set used for the evaluation of flag-based clause tagging in [HJN11]. From now on, we will refer to this set simply as SAT09. Being the SAT12 instance set too big for a feasible parameter selection, the solver has been tuned over a smaller subset consisting of 125 instances. The best configuration author could find over this set corresponds to the second best configuration author managed to find over the SAT09 instance set, thus author believes that this is a good configuration in general.

Instances have been chosen in order to obtain a set containing a fair number of satisfiable, unsatisfiable, easy, medium and hard instances. Hardness for each instance has been estimated by running 6 different solver configurations, three different ones with clause sharing (flag-based or position-based) disabled and three different ones with enabled clause sharing. For each configuration author has chosen:

- 5 instances where the wall-clock solving time was less than 100 seconds (easy instances)
- 5 instances where the wall-clock solving time was between 400 and 800 seconds (medium instances)
- 10 instances where the wall-clock solving time was between 800 and 2000 seconds (medium-hard instances)
- 10 instances where the wall-clock solving time was greater than 2000 seconds and less than the timeout (hard instances)
- 8 instances where the height of the partition tree has reached a level greater than 3 (memory consuming instances)

To the obtained result set (thus, devoid of duplicates) author added 10 instances that were not solved in any of the test runs performed up to that point. Finally, all those instances that could only be solved by those configurations where sharing was disabled, and the other way around, have been added as well.

5.4 Tests on SAT12

The experiments have been run in a multi-core setting using AMD Opteron 6274 CPUs with 2.2 GHz and 16 cores, so that we run 16 local solvers in parallel. The timeout for every instance is set to 1 hour (wall clock) and a total of 16 GB main memory is allowed for the parallel solver. Note that a parallel solver is intrinsically non-deterministic: running it several times over the same instance may result in different run times. Especially for satisfiable instances, it is known that by chance the solution is found much faster in the repetition of the run. However, in our specific

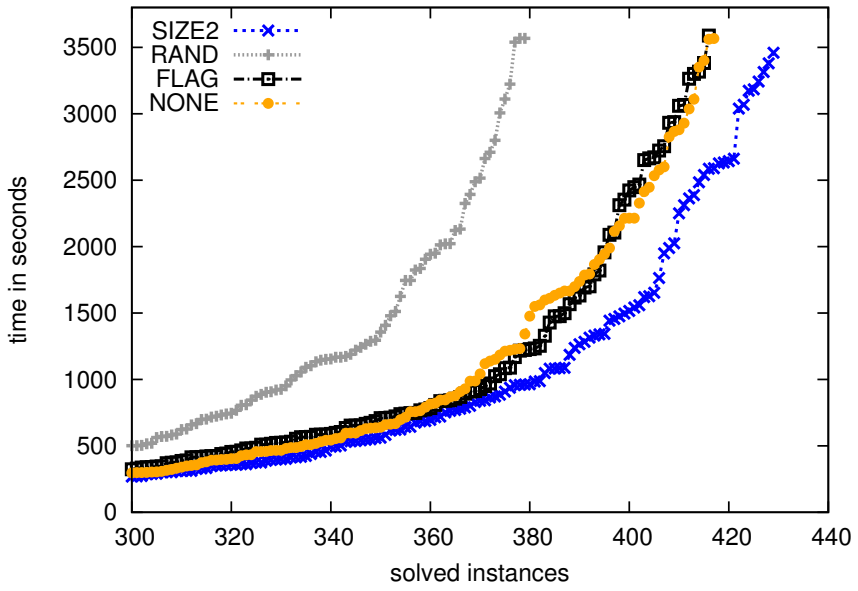

Figure 7: Solved instances and solving times of the four configurations on SAT12

case execution times have been quite stable, and thus the results here exposed are likely to be replicated.

The evaluation is performed over the following solver configurations:

1. `SIZE2`: position-based sharing restricted to unit and binary clauses
2. `RAND`: position-based sharing where any clause is shared with 5% probability
3. `FLAG`: flag-based sharing
4. `NONE`: no clauses are shared

Each of these configurations shares the same basic setting described in *Section* 5.1. Moreover, for configurations explotining clause sharing, sharing-delay parameter is set to one. This means that a clause C^p is put into the pool at position p as soon as it is learnt. In order to fill pools in a more homogeneous way, for our experiments

Table 2: Number of solved instances with 16 threads over SAT12 instance set

Approach	Solved	SAT	UNSAT	Median	AVG	CPU ratio	Score
`SIZE2`	**430**	239	**191**	**268.99**	377.397	11.4531	**78**
`RAND`	380	232	148	500.76	**374.445**	11.4951	-50
`FLAG`	417	234	183	324.29	378.969	11.396	30
`NONE`	418	**244**	174	296.73	383.785	**12.1013**	-58

Table 3: Clause sharing analysis with 16 threads over SAT12 instance set

Conf	SH LV 0	SH LV 1	Cur	Prev	Un	Bin	Dup	Glue	Tot
SIZE2	1021	4066	10608	1301	949	11395	18	12345	12345
RAND	19086	20446	111960	11760	35	689	1114	2188	134583
FLAG	4570	0	707	1566	222	4348	0	4570	4556
NONE	0	0	0	0	0	0	0	0	0

we artificially worsen the sharing level for SIZE2 and RAND (*dynamic levels* option): if a clause C should be sent to level k, then we send it to level $k' = k + log_2|C|$. The maximum size of each pool, for each configuration that makes use of sharing, is set to 10000 clauses.

5.4.1 Quantitative Analysis on Solved and Unsolved Instances

Table 2 gives various properties of the four configurations on the SAT12 instance set. For the whole set, the number of solved instances for both satisfiable (SAT) and unsatisfiable (UNSAT) formulae is given. SIZE2 slightly outperforms every other approach by solving at least 12 instances more. Surprisingly, this ranking gives a poor performance to previous work FLAG [HJN11].

With respect to median run time (Median), that is the time necessary to solve half of the whole instance set, SIZE2 still outperforms other configurations. This is not the case for the average run time of solved instances (AVG), where the best configuration is RAND. Even if at a first sight this could seem surprising, please notice that RAND solves far less instances than other approaches, meaning that difficult (with respect to run time) instances are likely not to be considered for the count of the average (since they are not solved). A confirmation of this fact can be observed in the cactus plot of *Figure* 7.

The cactus plot is a diagram with two axes: on the y-axis wall-clock times are reported, in seconds; on the x-axis, instead, the number of solved instances is given. For each element of the ordered sequence (ascending w.r.t. wall-clock time) of solved instances $(I_1, \ldots, I_n)$, a point $(j, wt(I_j))$, with $1 \leq j \leq n$, is "plotted" in the diagram, where $wt(I_j)$ is defined as the wall-clock time required to solve the instance I_j.

Table 4: Partition tree analysis with 16 threads over SAT12 instance set

Approach	#Nodes	#UnsolvedNodes	#SplitS	TreeHeight
SIZE2	45.8087	7.65225	0.500832	2.22795
RAND	33.8286	6.72879	0.667221	1.72712
FLAG	48.2662	8.3594	0.434276	1.95507
NONE	44.0582	8.18303	0.479201	1.91015

Table 5: Thread analysis with 16 threads over SAT12 instance set

Approach	Memory	CPUTime	RealTime	BlockedThreads
SIZE2	2448	17429	1289	96315
RAND	4348	17149	1555	6540
FLAG	2453	18356	1365	33
NONE	2698	18588	1363	0

For satisfiable instances, sharing no clauses seems to be the best opportunity, allowing the parallel solvers to diverse. On the other hand, for unsatisfiable instances the position-based sharing seems to be best. A good filter on clauses that can be shared is also important, as can be seen when SIZE2 is compared to RAND: SIZE2 solves more instances and the average run time per instance is almost the same. The *CPU ratio*, that is the total CPU time divided the total wall clock time, shows how many cores have been used in average to solve all the instances. Observe that this ratio is quite far from the ideal number (sixteen in these tests), but this does not depend on clause sharing. The splitter version used for these tests had some problems in re-allocating jobs every time new clauses were available. A more realistic measure, indeed, is obtained with the tests on SAT09 instances set by making use of an improved version of splitter (see *Section* 5.5). Nevertheless, one can still observe that the accesses to shared data structures do not alter this measure significantly: the value of configuration NONE, which does not share any clauses, is only slightly better than the other three configurations. It has been discussed whether only the number of solved instances is a good measure [Gel11]. A more careful ranking, which also takes solving times into account, gives a different picture. Column *Score* of *Table* 2 is based on work of [Gel11]. Every configuration is compared pairwise, and on each instance a "match" is played. If a configuration is faster on an instance, then it adds 1 to its current intermediate score, -1 otherwise. If two configurations are tie on a certain instance, then none of the configurations is added or subtracted points. Once all combinations have been compared, it is possible to create a matrix of intermediate scores. For details on how extract the final ranking scores, how in our table, from

Table 6: Scalability analysis 16 vs 4 threads over SAT12 instance set

	SAT		UNSAT		SAT run time		UNSAT run time	
Configuration	slower	faster	slower	faster	4-core	16-core	4-core	16-core
SIZE2	39	**203**	18	**175**	317.10	234.60	694.10	556.08
RAND	39	**195**	13	**136**	264.18	259.37	639.40	554.83
FLAG	48	**192**	21	**170**	293.53	235.51	636.49	562.41
NONE	18	**226**	16	**160**	296.16	235.91	603.22	591.20

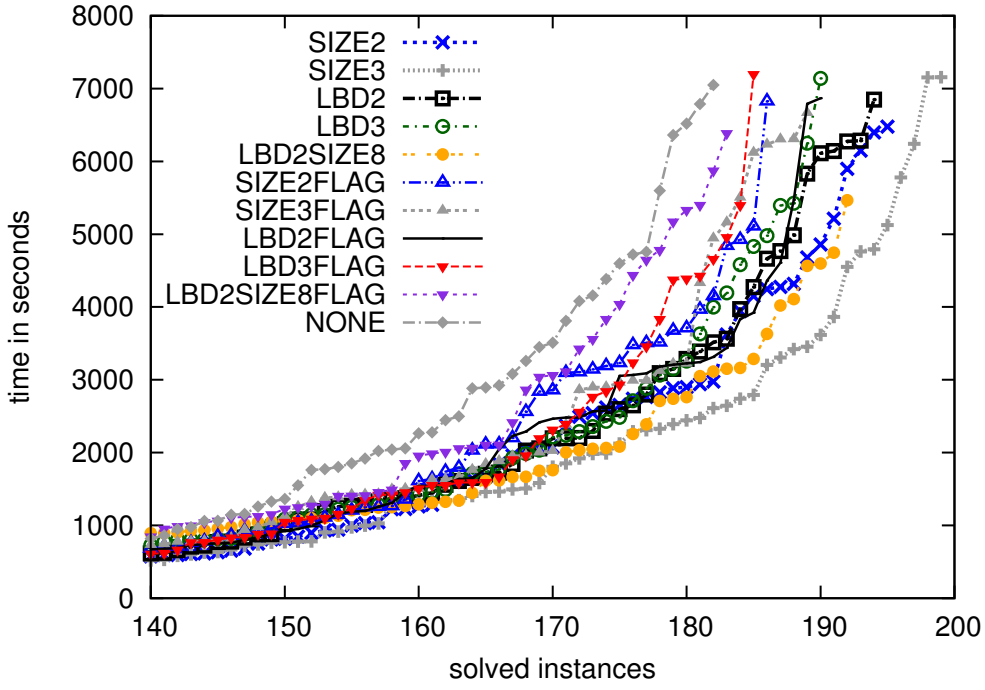

Figure 8: Solved instances and solving times of the four configurations

the matrix of intermediate scores please refer to [Gel11]. Regarding our experiments, the used noise value for ties has been set to the fixed value of 60 seconds. Now, `FLAG` shows the second best performance after `SIZE2`, which looks more like the expected evaluation. Furthermore, this ranking shows that the new approach outperforms the other configurations significantly. Comparing `SIZE2` directly with `NONE` the score is 37 to -37 points. Comparing to `FLAG`, `SIZE2` still wins with 13 to -13 points.

5.4.2 Clause Sharing Analysis

We analyzed how many clauses have been shared, and furthermore recorded the subtree where these clauses are valid. *Table* 3 shows several details regarding clause sharing, namely: the average number of clauses per instance that are sent to the pool of level zero (SH LV 0); the average number of clauses per instance that are sent to pools of level one (SH LV 1); the average number of clauses per instance that have been tagged with the same positions of those nodes where they have been learnt (Cur); the average number of clauses per instance that are learnt at level n and tagged with level $n-1$ (Prev); the average number of shared clauses per instance that are unit (Un); the average number of shared clauses per instance that are binary (Bin); the average number of shared clauses per instance whose ldb score is less or equal than two (Glue); the average number of shared clauses per instance (Tot). Obviously, `NONE` does not share any clause. The configuration `FLAG` shares only clauses, if they are valid for the whole formula. Therefore, all 6557 clauses have been sent to the storage

Table 7: Number of solved instances with 8 threads over SAT09 instance set

Approach	Solved	SAT	UNSAT	Median	AVG	CPU ratio	Score
SIZE2	196	82	**114**	**632.74**	753.23	6.66111	13
SIZE3	**200**	**87**	113	634.85	741.661	6.46252	175
LBD2	195	84	111	772.67	783.159	6.47885	67
LBD3	191	86	105	816.26	711.404	6.26359	110
LBD2SIZE8	193	87	106	955.28	672.434	5.92167	57
SIZE2FLAG	187	84	103	848.28	668.747	6.46023	-125
SIZE3FLAG	190	84	106	811.00	744.533	6.5868	-30
LBD2FLAG	191	85	106	755.36	689.209	6.56635	-125
LBD3FLAG	186	83	103	801.26	633.958	6.39582	21
LBD2SIZE8FLAG	184	83	101	1062.40	725.946	5.9966	-240
NONE	183	82	101	1064.84	771.522	**6.79761**	-278
LBD2SIZE8SH32	187	84	103	869.62	**607.565**	6.17128	**280**

of the initial formula. Sharing clauses randomly with 5 % probability results in the most shared clauses, namely 209199 in RAND. Note that only 14 % of all clauses are shared among all nodes in the partition tree. Another 15 % of the clauses are sent to pools of level 1. All remaining clauses are sent lower in the tree and thus shared among fewer nodes. Restricting shared clauses to binary and unit clauses in SIZE2 leads to less shared clauses, namely only 17202. Similarly to RAND, only a small fraction (8 %) of these clauses is sent to the root of the partition tree. Note that both SIZE2 and RAND share binary clauses not at the root node, so that the number cannot be easily compared to FLAG. Still, SIZE2 shares more clauses than FLAG in total, and also results in a higher performance. Finally, the number of duplicates in the pools is relatively low for every configuration. A reason for this is because the pools are relatively small, as in *Section* 5.5 with an increased pool picture will slightly change. Summarising, it is shown that our new sharing approach can share more clauses than previous approaches. Also, these clauses are useful for the search since the solver performs better than the previous one. However, configuration RAND shows that simply sending any clause without a good filter results in a degradation of performance. This is in line with the expected behaviour and previous research on clause sharing. The used size restriction seems to be a good filter heuristic, or at least better than not sharing clauses at all. However, we anticipate that with a better semaphores access policy (and with less threads) better results can be achieved by allowing more clauses to be shared (see *Section* 5.5).

5.4.3 Partition Tree and Resource Consumption Analysis

Interesting insight is given from *Tables* 4 and 5, describing respectively the shape of the partition tree and the consumption of resources (memory and time) from threads.

Table 8: Clause sharing analysis with 8 threads over SAT09 instance set

Conf	SH LV 0	SH LV 1	Cur	Prev	Un	Bin	Dup	Glue	Tot
SIZE2	3063	1031	5403	2038	863	7763	27	8626	8626
SIZE3	16422	4166	20250	9074	751	6224	14967	14137	42931
LBD2	10010	3025	12947	6215	848	6595	3584	25855	25855
LBD3	21167	8118	30718	12797	488	3421	4608	10642	53097
LBD2SIZE8	32399	19157	92163	23145	455	3380	4170	9068	128062
SIZE2FLAG	3108	0	664	1239	177	2930	32	3108	3108
SIZE3FLAG	15175	0	2060	4188	181	2852	6388	6587	15175
LBD2FLAG	10862	0	1547	3136	166	2293	3207	10862	10862
LBD3FLAG	22037	0	4904	8530	153	1961	3483	5664	22037
LBD2SIZE8FLAG	34064	0	8626	13695	162	1998	2699	4996	34063
NONE	0	0	0	0	0	0	0	0	0
LBD2SIZE8SH32	30932	18962	100377	22311	443	3298	4302	9000	134253

Regarding *Table* 4, the information provided is: the average size of the partition tree per instance in terms of number of nodes (#Nodes); the average number of nodes per instance for which the timeout has been reached without solving the assigned task (#UnsolvedNodes); the average number of nodes per instance that have been solved while attempting to split them, either by a splitter or solver thread (#SplitS); the average height of the partition tree per instance (TreeHeight).

First, notice how the average height of a partition tree per instance is quite low. This is both because SAT12 set contains several easy instances and because the cardinality of the splits is quite high, namely 8. Moreover, it seems that sharing a huge number of clauses tends to produce smaller partition trees, as for configuration `RAND` the average number of nodes per instance is 34 against 44 of `NONE`. An explanation for this fact comes from the observation that the average number of solved nodes per instance increases when `RAND` is used, preventing new splits from being created because of a timeout. This is a good news as by creating less nodes the overhead introduced by the splitting phase is reduced.

Finally, observe that the number of nodes solved during the splitting phase is in approximation only the 1% of the average number of solved nodes per instance. This is not surprising as a splitter thread usually has a very short life; consequently, the chances of solving a node during this elapse of time are quite low.

Table 5 provides information about the resources used by the parallel solver: Memory is the average number of megabytes per instance required by the whole parallel procedure; CPUTime is the average cpu time spent on each instance, measured in seconds; RealTime is the average wall-clock time spent on each instance, measured in seconds; BlockedThreads is the average number of times per instance that threads tried to access pools under a write lock (since we are using read-write locks, encoun-

Table 9: Partition tree analysis with 8 threads over SAT09 instance set

Approach	#Nodes	#UnsolvedNodes	#SplitS	TreeHeight
SIZE2	50.5205	6.96575	1.0411	2.20205
SIZE3	46.3322	6.40411	1.08219	2.11644
LBD2	49.4829	6.84247	1.05479	2.16781
LBD3	30.7158	4.38014	1.08904	1.92808
LBD2SIZE8	27.2329	3.92466	1.19521	1.91781
SIZE2FLAG	32.387	5.46575	0.84589	2.07192
SIZE3FLAG	39.2055	6.15753	0.842466	2.00342
LBD2FLAG	43.0856	6.59247	0.938356	1.9726
LBD3FLAG	31.4795	5.13356	0.982877	2.03767
LBD2SIZE8FLAG	24.161	4.17808	0.825342	1.86986
NONE	45.0685	6.92466	0.842466	2.01027
LBD2SIZE8SH32	25.8733	3.61301	1.15068	1.83562

tering a lock means that another thread is already pushing clauses on the same pool).

Unfortunately, memory usage becomes an important issue when the number of shared clauses increases. Indeed, observe that RAND requires twice as memory as NONE. There are at least two explanations for this increased memory usage:

1. pools require memory to allocate clauses
2. solvers pulling clauses from a pool tend to have bigger learnt databases

The second problem can be tackled by introducing a modified cleaning strategy that takes into account the fact that some clauses are not learnt locally but received from other solvers [AHJ+12a].

The first problem, instead, can be alleviated by deleting those pools that cannot be accessed by any thread (e.g., pools on solved nodes). In *Section* 5.5 we perform tests with an improved version of the solver that presents this pools deletion mechanism; the memory overhead introduced by this improved solver is dramatically lower than the overhead of the original version.

5.4.4 Scalability Analysis

To check whether for future multi-core architectures the approach will scale further, the four solver configurations are run also with a restriction to 4 cores and we measure the run time and number of solved instances again. For instances that could be solved with 4 or 16 cores we give the number of instances that can be solved faster with one of the two approaches in *Table* 6. Column *slower* (resp. *faster*) counts the total number of instances where the solver with 16 threads was slower (resp. faster) than its counterpart with 4 thread. Furthermore, the average run time per solved instance

Table 10: Thread analysis with 8 threads over SAT09 instance set

Approach	Memory	CPUTime	RealTime	BlockedThreads
SIZE2	617	3527	2723	60957
SIZE3	615	3227	2643	104730
LBD2	617	3415	2673	86268
LBD3	612	3057	2588	77204
LBD2SIZE8	666	2885	2560	122744
SIZE2FLAG	599	2932	2748	18834
SIZE3FLAG	640	3225	2685	80785
LBD2FLAG	604	3098	2716	76295
LBD3FLAG	608	2736	2654	70391
LBD2SIZE8FLAG	609	3191	2757	98067
NONE	**556**	3398	2918	**0**
LBD2SIZE8SH32	642	**2589**	**2571**	11258

is compared (columns SAT run time and UNSAT run time). The data shows that most of the instances benefit from additional resources. The run time comparison shows that most of the times using 16 cores instead of 4 cores results in a higher performance of the solver. Only a few instances are slower. A similar picture is also presented when the average run times are compared. For all configurations the average run time decreases when more resources are used. These results are in line with the results of [HM12b], where no clause sharing was used.

5.5 Tests on SAT09

Tests are performed on the same machine used for tests on SAT12 but, instead of 16 threads, the parallel sat solver is allowed to work at with eight threads, only (that is the same number of threads usually allowed in SAT competitions). Another difference is that the timeout is extended from 1 hour to 2 hours. The basic configuration of splitter is the same as for the experiment on SAT12, but the implementation is slightly different as some minor bugs were fixed resulting in a better usage of memory and cpu resources (see *Sections* 5.5.1 and 5.5.3).

Due to the improvement that permits to delete useless pools of shared clauses, discussed in 5.4.3, and from the observation that the average number of shared clauses per instance is 134583 for the RAND configuration (see *Table* 3), pools are allowed a maximum size of 450000 clauses, instead of the limit 10000 used for the tests on SAT12. Consequently, the dynamic-level option is turned off since removal of useful clauses from pools is more unlikely. Experimental results confirmed this intuition, as for SIZE2 configuration dynamic-level option is worse than having it disabled (187 solved instances against 196).

Being SAT09 set of instances smaller than SAT12 (292 against 600), we managed

Table 11: Efficiency and speedup analysis for 8 threads over SAT09 instance set

Configuration	$E > 1$		$T_s < T_p$		$E \geq 0.5$		#solvedBoth
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	
SIZE2	**15**	**8**	**4**	15	17	10	169
SIZE3	13	6	4	21	15	11	162
LBD2	14	7	6	**14**	**23**	11	168
LBD3	13	8	4	20	13	15	161
LBD2SIZE8	10	4	9	18	18	14	164
SIZE2FLAG	14	7	6	23	18	6	163
SIZE3FLAG	15	6	5	20	11	10	162
LBD2FLAG	13	6	5	17	20	13	165
LBD3FLAG	10	5	6	22	20	13	160
LBD2SIZE8FLAG	11	5	9	27	14	10	160
NONE	14	4	4	30	17	7	161
LBD2SIZE8SH32	14	8	8	20	19	**17**	160

to perform more accurate tests than on SAT12 and for this discussion we selected the following meaningful configurations:

1. `SIZE2`: position-based sharing restricted to unit and binary clauses
2. `SIZE3`: position-based sharing restricted to unit, binary and ternary clauses
3. `LBD2`: position-based sharing restricted to clauses of $lbd \leq 2$
4. `LBD3`: position-based sharing restricted to clauses of $lbd \leq 3$
5. `LBD2SIZE8`: position-based sharing restricted to clauses with $lbd \leq 2$ or size ≤ 8
6. `SIZE2FLAG`: flag-based sharing restricted to unit and binary clauses
7. `SIZE3FLAG`: flag-based sharing restricted to unit, binary and ternary clauses
8. `LBD2FLAG`: flag-based sharing restricted to clauses of $lbd \leq 2$
9. `LBD3FLAG`: flag-based sharing restricted to clauses of $lbd \leq 3$
10. `LBD2SIZE8FLAG`: flag-based sharing restricted to clauses with $lbd \leq 2$ or size ≤ 8
11. `NONE`: solver without clause sharing
12. `LBD2SIZE8SH32`: as (5), but shared clauses can be pushed only every 32 conflicts

5.5.1 Quantitative Analysis on Solved and Unsolved Instances

Cactus plot in *Figure* 8 shows how `SIZE3` is the best configuration on this set of instances, as it slightly outperforms the runner-up configuration `SIZE2`. Observe that `LBD2SIZE8SH32` is purposely left out of the cactus plot, but it will be discussed in the tables coming next. Remember that `SIZE2` was the best configuration in experiments over SAT12 set of instances. The reason why `SIZE3` was not tested on that set of instances is because it performed worse than `SIZE2` on the tuning selection for SAT12 (see *Section* 5.3). From the moment that the last version of the solver can deal with more shared clauses, because of some minor fixes in implementation, most likely `SIZE3` would perform better than `SIZE2` on SAT12 as well. This might be interesting for future work.

Table 7 reports data regarding the number of solved and unsolved instances, median run time, average run time and CPU ratio. For a description of the column names please refer to *Section* 5.4.1.

The first thing to notice is that, due to a better implementation, the results are more convincing than the ones reported for SAT12: `SIZE3` solves 17 instances more than `NONE`. Translated into percentage, position-based sharing allowed to obtain an improvement up to 5.82% in terms of number of solved instances against the standard non-sharing solver, whereas in SAT12 tests the best sharing configuration only managed to achieve an improvement of 2.7%.

As for SAT12, also in this case we observe how sharing clauses helps especially in case the formula is unsatisfiable. Indeed, `SIZE3` solves 13 more unsatisfiable instances than `NONE`; however, notice that improvements on satisfiable formulae are obtained as well, as `SIZE3` solves 4 more satisfiable instances than `NONE`.

In general, it seems that as more clauses are shared the solver becomes faster. Indeed, configuration `LBD2SIZE8`, that shares the highest number of clauses (see *Table* 8), looks to be the fastest configuration (*Figure* 8) even though it cannot solve as many instances as `SIZE3`. A reason for this can be observed in the bad CPU ratio for this configuration: due to communication costs, almost 1 core is lost. Configuration `LBD2ORSIZE8SH` tries to overcome the problem, by setting the sharing-delay parameter to 32 conflicts. This has two consequences: 1) a lock is requested every 32 conflicts instead of every conflict and 2) each time a thread locks a pool, it is allowed to push more than 1 clause (up to 32) without releasing the lock. As a result, threads lock the pools less often but for a longer time (see column "BlockedThreads" in *Table* 10). Observe how with this small change we gain around half a core, and the solver becomes faster in terms of median and average time. Nevertheless, `LBD2SIZE8` solves 4 instances more than `LBD2SIZE8SH`. This could depend from the lack of robustness of divide-and-conquer parallel CDCL solvers: repetitions of the test for the same configuration will lead to slightly different results. A hint that is indeed the case comes from the fact that `LBD2SIZE8` solves 3 satisfiable instances more than `LBD2SIZE8SH`.

As `NONE` is the worst configuration (in terms of the number of solved instances), every flag-based configuration is better than non-sharing at all. This confirms results in [HJN11], displaying a different picture from the analysis performed in *Section* 5.4.1. Still, observe that every flag-based configuration is worse than its position-based coun-

terpart. One might think that by allowing flag-based approaches to share more clauses this situation might change. Comparison of `LBD2SIZE8FLAG` and `LBD2SIZE8`, however, tells us that this is not the case; even when the number of shared clauses becomes high, the performance of flag-based still cannot match the one of position-based. Indeed, `LBD2SIZE8FLAG` is one the worst configuration both in terms of solved instances and in terms of score, whereas `LBD2SIZE8` is among the fastest and it solves 9 instances more than its flag-based counterpart. This is reasonable, since safe clauses are all kept in a single pool: each thread will try to take a lock on that pool. The picture will become clearer when we will analyse the details of the clause sharing for each configuration, in *Section* 5.5.2.

Another interesting information is that *size-based* filters on shared clauses seem to be better than *lbd-based* ones. This is not surprising, as lbd is not a global measure but it is always relative to the solver where the lbd is evaluated. Thus, a clause with a good lbd score on a solver is likely to have a worse lbd score on another solver: receiving such a clause for this last solver would be pointless.

Finally, observe how scores from careful ranking give a slightly different picture than the classical analysis based on the number of solved instances. `LBD2SIZE8SH`, now, dominates every other approach with a score of 280, whereas `SIZE3` places only second with a score of 175. The worst configuration, even with this ranking, remains `NONE` with an awkward score of -278.

5.5.2 Clause Sharing Analysis

Table 8 provides information regarding clause sharing. Description of the column names for this table can be found in *Section* 5.4.2.

Observe that, for every flag-based configuration, its position-based counterpart shares around thrice as many clauses. This conforms to what observed over the SAT12 set of instances. A difference, instead, is in the number of duplicates in the shared pools: for configurations as `SIZE3`, the number of times a thread tries to push a clause that is already present in the pool becomes significant. However, there seems to be not a direct correlation between the number of shared clauses and the number of duplicates, as configurations like `LBD2SIZE8SH` or `LBD2SIZE8` present a small amount of duplicates when compared to the total number of shared clauses. This is a very interesting behaviour, that deserves to be studied. Most probably this is a consequence of some common underlying structure for instances of SAT09 set, but we do not have enough data to confirm (or refute) this hypothesis.

Without surprise, configurations with the highest number of shared clauses are `LBD2SIZE8SH` and `LBD2SIZE8`. Furthermore, observe that the average number of clauses shared per instance by `LBD2SIZE8SH` is comparable with the number of clauses shared per instance by `RAND` over the SAT12 instances set. By exploiting this similarity, in *Section* 5.5.3 we will estimate the impact on memory consumption when deleting non-accessible pools.

Consider the configuration `LBD2SIZE8SH`. Observe that only 23% of all the clauses are safe, as they are sent to the pool of level zero, and all the others are unsafe. Of these unsafe clauses, a 14% is sent to pools of level one, whereas the remaining 73%

is shared downer in the tree.

Percentages for `LBD2SIZE8SH`, similar to the ones for `LBD2SIZE8`, might explain why configuration `LBD2SIZE8` is one of the fastest, and `LBD2SIZE8FLAG` one of the slowest: the CPU ratio for both instances is similar (see *Table* 7), but the number of clauses shared by `LBD2SIZE8FLAG` is significantly lower than the number of clauses shared by `LBD2SIZE8` (34063 against 128062 clauses). This means that, for flag-based clause tagging, communication overhead on equal number of shared clauses is significantly higher than its position-based counterpart. Observe that the problem might be alleviated by providing more pools for storing safe clauses. However, this has not been tested and even work in [HJN11] uses a single pool.

There seems to be a correlation, instead, between the average number of unit and binary clauses per instance and the filter used on shared clauses: the stricter the filter, the higher the number of shared small clauses. For example, `SIZE2` shares in average 7763 binary clauses per instance, whereas less restricting configurations like `LBD2SIZE8` or `LBD2SIZE8SH` share only 3380 and 3298 binary clauses respectively. An even more drastic decrease can be observed when considering unit clauses. From these facts, it seems that sharing longer clauses somehow prevents receiving solvers from learning "short" clauses. However, a more plausible and natural explanation can be given when analysing the shape of the partition tree generated by these configurations, as we will do in *Section* 5.5.3.

5.5.3 Partition Tree and Resource Consumption Analysis

Table 9 presents information regarding the shape of the partition tree for each configuration. Reader is reminded that column names for this table are described in *Section* 5.4.3.

Observe that information in this table is compatible with information in *Table* 4. For example, the average number of created nodes is around 40 for both tables. Furthermore, the inverse proportion between the total number of shared clauses and the number of nodes in the tree seems to be confirmed. Observe that this might explain why less short clauses are learnt when the total number of clauses increases (see *Section* 5.5.2). Short clauses, indeed, are likely to be learnt at the initial stages of the search process of a sequential CDCL solver [HJS09a]. A partition tree with more nodes, thus, will present more often the situation where a thread is at the initial stage of the search over a certain node. Consequently, it will present more often a situation that is favourable to the sharing of short clauses.

A different picture from the one offered by the test over the SAT12 set of instances, instead, is presented by the memory analysis table (*Table* 10). Column names for this table are explained in *Section* 5.4.3. One can observe how the impact of deleting unreachable pools is significant in terms of memory consumption: `RAND` configuration on SAT12 tests uses an amount of memory per configuration that is double the amount of a non-sharing configuration (see *Section* 5.4.3). Observe that, even though `RAND` and `LBD2ORSIZE8` share a similar total number of clauses, the memory consumption of `LBD2ORSIZE8` is only a 15% more than the memory consumption of no sharing. However, as already explained in *Section* 5.5.2, a direct comparison is not possible as

the average size of the clauses shared in `RAND` is bigger than the average size of the clauses shared in `LBD2ORSIZE8`. Nevertheless, consider that the maximum size of the pools for shared clauses used in tests on SAT12 instance set was only 10000, whereas in SAT09 tests pools are allowed to grow up to a size that is 45 times bigger. With these considerations, it is plausible to assume that memory consumption for `RAND` and `LBD2ORSIZE8` configurations are comparable. Still, author is aware of the fact that, in order to precisely quantify this memory gain, further tests are necessary.

Another interesting measure is the average number of times per instance that threads are blocked. Observe how this measure for configuration `LBD2SIZE8SH` is drastically lower than the one for `LBD2SIZE8` (11258 against 122744). This is expected, as locks requests in `LBD2SIZE8SH` are not performed each time a clause is learnt, but every 32 learnt clauses. As already discussed in *Section* 5.5.2, this translates in an improved performance in terms of exhibited CPU ratio.

5.5.4 Efficiency and Speedup Analysis

We conclude the analyses on SAT09 set of instances by performing an efficiency and speedup study. As T_s, we consider the wall clock run time of MiniSat over SAT09 instances. Results are shown in *Table* 11. Sporadic super-linear speedups can be obtained for both satisfiable and unsatisfiable formulae ($E > 1$), but speedups on unsatisfiable instances seem to increase when more clauses are shared (e.g., configuration `LBD2SIZE8`). Unfortunately, slowdowns are possible ($T_s < T_p$), especially on unsatisfiable instances and when clause sharing is disabled. Column $E \geq 0.5$ counts, for each configuration, how many instances have been solved with a "good" efficiency. Unfortunately, the numbers are low, meaning that in average the efficiency is always quite far from 1. Last column reports the number of instances that have been solved from both MiniSat and the configuration under exam. Only over those instances, indeed, speedup and efficiency analysis can be performed.

6 Conclusion and Future Work

This work presented a new position-based clause sharing technique that allows to share clauses for subsets of nodes of a parallel iterative partitioning SAT solver.

We formally proved correctness of the newly introduced approach, and we showed how this proof can be adapted into a correctness proof also for the previous state-of-the-art approach, namely flag-based clause tagging [HJN11]. This is the first proposed proof for flag-based approach.

We provided a rigorous empirical evaluation and analysed our approach (and its competitors) in terms of performance, resources consumption, nature of the information that is shared and ability to scale when more resources are added. Our tests display the effectiveness of position-based clause tagging: it allows more clauses to be shared, in comparison with flag-based clause tagging, without introducing significant communication overhead. Moreover, we showed how flag-based approach (as presented in [HJN11], that is with a single centralised pool for safe clauses) cannot share the same amount of clauses that are shared by a position-based counterpart without introducing a significant communication overhead.

These additional "cheap" (in terms of communication overhead) shared clauses effectively contribute to the search, as position-based clause tagging significantly outperforms approaches where clause sharing is not performed or clause sharing is performed by exploiting the previous flag-based technique.

Moreover, we observed that the approach is scalable as when resources are added to the parallel procedure, the overall performance increases as well.

Results here obtained are coherent with current state of research, as several other authors reported how clause sharing can effectively increase the performance of a parallel CDCL based SAT solver. With this respect, our empirical evaluation shows how conclusions in [HJN11] are valid even when the underlying architecture is multi-core rather than a network grid.

Future work could improve the implemented sharing mechanism further. By using a dynamic filter on clauses that can be shared, instead of a fixed-size-or-lbd one, important improvements are likely to be achieved [HJS09a]. Moreover, a selection of those threads which send "important" clauses can decrease communication overhead and allow more clauses to be shared [LHJS12]. Furthermore, once acquired by a receiver solver, shared clauses should be treated with a dedicated cleaning policy rather than considering them as simple (sequentially) learnt clauses [AHJ$^+$12a]. All these extensions have been successfully applied to portfolio state-of-the-art procedures, and are compatible with our clause sharing mechanism. Another improvement, not related to portfolio solvers, might come by performing a conflict analysis over solved unsatisfiable nodes: it the empty clause can be obtained through a resolution derivation involving safe clauses only, then the whole parallel procedure can return the unsatisfiable answer without need to wait for all other nodes in the partition tree. Of course, this idea can be refined further by exploiting the potentiality of position-based clause tagging.

Additionally, improvements of the basic splitter algorithm could result in a competitive parallel SAT solver when enhanced with position-based clause sharing. Possible

improvements include the analysis and evaluation of more sophisticated search space partitioning functions, or improvements on the underlying sequential solver, as for as for example restarts and advanced search direction techniques.

References

[AAA+08] Mikko Alava, John Ardelius, Erik Aurell, Petteri Kaski, Supriya Krishnamurthy, Pekka Orponen, and Sakari Seitz. Circumspect descent prevails in solving random constraint satisfaction problems. *Proceedings of the National Academy of Sciences*, 105(40):15253–15257, October 2008.

[AHJ+12a] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 200–213, Berlin, Heidelberg, 2012. Springer-Verlag.

[AHJ+12b] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Penelope, a parallel clause-freezer solver. In *proceedings of SAT Challenge 2012: Solver and Benchmarks Descriptions*, pages 43–44, Lens, may 2012.

[ALMS11] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 188–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[Arn09] Holger Arnold. A linearized dpll calculus with clause learning. 2009.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[AS12] Gilles Audemard and Laurent Simon. Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (system description). In *Pragmatics of SAT 2012(POS'12)*, jun 2012.

[BBD+12] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors. *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2012. ISBN ISBN 978-952-10-8106-4.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[BS96] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Manage. Sci.*, 35(2):164–176, February 1989.

[CW03] Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[Dij68] Edsger W. Dijkstra. The structure of "THE" multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, June 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[ES05] N. Een and N. Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition, 2005.

[FDH05] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2005.

[Gel11] Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *SAT*, pages 317–328, 2011.

[GFS08] Luís Gil, Paulo Flores, and Luís M. Silveira. PMSat: a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.

[GHM+12] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving periodic event scheduling problems

with sat. In He Jiang, Wei Ding, Moonis Ali, and Xindong Wu, editors, *IEA/AIE*, volume 7345 of *Lecture Notes in Computer Science*, pages 166–175. Springer, 2012.

[Goe10] Asvin Goel. A column generation heuristic for the general vehicle routing problem. In Christian Blum and Roberto Battiti, editors, *LION*, volume 6073 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2010.

[GSH08] Chu Georey, Peter J. Stuckey, and Aaron Harwood. PMiniSat - A parallelization of MiniSat 2.0. 2008.

[HHLB13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *Proc. of LION-7*, 2013. To appear.

[HJN06] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving sat in grids. In *Proceedings of the 9th international conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 430–435, Berlin, Heidelberg, 2006. Springer-Verlag.

[HJN10] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010.

[HJN11] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based sat solving with iterative partitioning and clause learning. In Jimmy Ho-Man Lee, editor, *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2011.

[HJS09a] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[HJS09b] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.

[HKWB12] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference 2011*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65, 2012. Best Paper Award.

[HM12a] Antti E. J. Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 214–227, Berlin, Heidelberg, 2012. Springer-Verlag.

[HM12b] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 214–227, 2012.

[HMN+11] S. Hölldobler, N. Manthey, V.H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel SAT-solvers. In I. Wasito et.al., editor, *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011. ISBN 978-979-1421-11-9.

[HMS10] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware sat solvers. In Christian Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HW12] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel sat solving. In *Proceedings of AAAI*. AAAI Press, 2012.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. In *In Proceedings of Workshop on Theory and Applications of Satisfiability Testing (SAT'2001*, pages 205–211. Elsevier Science Publishers, 2001.

[Kas90] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *AI*, 45(3):275–286, Oct 1990.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In *IN ECAI-92*, pages 359–363. Wiley, 1992.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 343–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[LHJS12] Nadjib Lazaar, Youssef Hamadi, Said Jabbour, and Michèle Sebag. Cooperation control in Parallel SAT Solving: a Multi-armed Bandit Approach. Rapport de recherche RR-8070, INRIA, September 2012.

[LSB07] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.

[Man11] Norbert Manthey. Parallel SAT Solving - Using More Cores. In *Pragmatics of SAT(POS'11)*, 2011.

[MML10] Ruben Martins, Vasco Manquinho, and Ines Lynce. Improving search space splitting for parallel sat solving. In *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, ICTAI '10, pages 336–343, Washington, DC, USA, 2010. IEEE Computer Society.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[MSLM] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153.

[PD10] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *J. Autom. Reason.*, 44(3):277–301, March 2010.

[PKA+06] Stephen Plaza, Ian Kountanis, Zaher Andraus, Valeria Bertacco, and Trevor Mudge. Advances and Insights into Parallel SAT Solving. In *IWLS 2006: 15th International Workshop on Logic and Synthesis*, June 2006.

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[ZBH96] HANTAO ZHANG, MARIA PAOLA BONACINA, and JIEH HSIANG. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4–6):543 – 560, 1996.

[Zha97] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[ZMM01] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 03. April 2013

Your Name