TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Artificial Intelligence Knowledge Representation and Reasoning

Sharing Information in Parallel Search with Search Space Partitioning

Davide Lanti Norbert Manthey

KRR Report 13-01

Mail to
Technische Universität Dresden
01062 Dresden

Bulk mail to
Technische Universität Dresden
Helmholtzstr. 10
01069 Dresden

Office
Room 2006
Nöthnitzer Straße 46
01187 Dresden

Internet
http://www.wv.inf.tu-dresden.de

Sharing Information in Parallel Search with Search Space Partitioning

Davide Lanti and Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

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, for example the well researched SAT problem. In this paper we show how information can be shared in a parallel SAT solver that relies on search space partitioning via iterative partitioning. With improved clause sharing, 12 more instances out of 600 instances can be solved. We further show that the new approach also results in a higher performance with respect to alternative rankings.

1 Introduction

Search problems arise from various domains, ranging from small logic puzzles over scheduling problems like railway scheduling [1] or vehicle routing [2] to large job shop scheduling problems [3]. As long as the answers to these problems need not to be optimal, these problems can be translated into a constraint satisfaction problem [4], or into satisfiability testing (SAT) [5]. Usually, SAT instances are solved with DPLL [6] style SAT solvers [7]. The best improvement of the DPLL algorithm is learning information during search [8,9], which is called *clause learning* where *learnt* clauses are added to the formula. Solving search problems with SAT usually results in increased performance, because SAT is a highly researched area and available SAT solvers are very sophisticated. For example scheduling railway trains has been improved by a speedup up to 10000 compared to the state-of-the-art domain specific solver [1]. For the optimization variant of scheduling problems, the *maximum satisfiability problem* (MaxSAT) [10] can be used. State-of-the-art MaxSAT solvers rely on SAT as a back end, such that the search for an optimal solution inside these solvers heavily depends on the search process of the underlying SAT solver [11,12].

With the advantage of parallel computing resources, for example multi-core CPUs, also parallel SAT solvers have been developed [13,14,15]. Since modern CPUs start to receive more and more cores, scalability studies of parallel SAT solvers become important. Parallelizing the SAT solving algorithm itself has been studied in [16]. However, the approach studied there does not scale beyond two cores, because structured SAT algorithms are very similar to a depth-first search, which in the worst case cannot be parallelized [17], and also practical instances do not provide enough parallelism.

Other parallel SAT solving approaches combine the search of different solver incarnations. Hyvärinen et.al. [18] study scalability of the three most common parallelization approaches, namely parallel portfolio search, plain search space partitioning, and iterative search space partitioning. Portfolio search gives the whole search space to different solvers and runs them in parallel. The fastest solver returns the result to the user. This technique is widely used [13,14] and yields a good performance for small numbers of cores. Plain partitioning search first divides the search space into partitions and then gives each partition to a solver. Finally, all solver results are combined and an answer is given. Unfortunately, this approach suffers from a theoretical slow down [18]. An improvement is iterative search space partitioning: the initial problem is solved with a sequential solver and limited resources, e.g. limited run time. Additionally, the search space is divided into sub-spaces. If solving a sub-space is not finished within the given resources, this sub-space is partitioned again. Comparing the presented three approaches, Hyvärinen et.al. conclude that iterative search space partitioning gives the best scalability. However, this technique is not researched much. While for portfolio solvers there exists many improvements, for example sharing learned information according to some filter heuristics [19,13] or controlling the diversification and intensification among the solvers [20], iterative search space partitioning received little attention; for a grid implementation of the parallel solver, only a study on how to divide the search space and on limited sharing has been done [21,22]. As for portfolio solvers [13,23], Hyvärinen et.al report that in average even this limited sharing results in a speedup.

In this paper we present an improved clause sharing mechanism for the parallel iterative partitioning approach. To divide the search space of a formula into sub-spaces, Hyvärinen et.al. add so called *partition constraints* to the formula [22]. Only learnt clauses that do not depend on these partition constraints are shared with other solvers, and clauses are only sent after a solver finished to work on a sub-space. A computing grid is used as underlying computing resources. To further improve the scalable parallel algorithm, we contribute a more general sharing mechanism for the iterative partitioning approach and evaluate this scheme on multi-core CPUs. First, we share learnt clauses that also depend on partition constraints, but send them only to solvers where these clauses are valid. Additionally, learnt clauses are sent during search so that other solvers may benefit immediately.

Our evaluation reveals interesting insights. First, sharing clauses introduces almost no overhead in computation. Furthermore, the performance of the overall search is increased. Two different rankings [24,25], the first being used in international competitions and the latter being more stable than the first ranking, show that the new approach has a higher performance compared to sharing no clauses or restricted sharing. One of the reasons for this improved behavior is that the number of shared clauses increases, strengthening the cooperation among the parallel running solvers. Finally, the approach scales with more cores; when the number of cores is increased from 4 to 16, the performance of the overall system also improves.

After giving more detailed preliminaries on SAT solving in Section 2, we show how the iterative partitioning approach can be improved by sharing learned information in Section 3, and afterwards, we evaluate our approach in Section 4. Finally, we conclude and give an outlook in Section 5.

2 Preliminaries

After providing the necessary notations for satisfiability testing, we show how the depth-first like search algorithm to solve SAT instances is enhanced with clause learning. Finally, we discuss related work on parallel SAT solving.

2.1 Satisfiability Testing

Let V be a finite set of Boolean variables. The set of *literals* $V \cup \{\overline{x} \mid x \in V\}$ consists of positive and negative Boolean variables. A *clause* is a finite disjunction of literals and a *formula* (in *conjunctive normal form* (CNF)) is a finite conjunction of clauses. We sometimes consider clauses and formulae as sets of literals and sets of clauses, respectively, because duplicates can be removed safely. A *unit clause* is a clause that contains a single literal. We denote clauses with square brackets and formulae with angle brackets, so that $((a \vee b) \wedge (\overline{a} \vee c \vee \overline{d}))$ is written as $\langle [a, b], [\overline{a}, c, \overline{d}] \rangle$. Furthermore, we define two helper functions: $\mathsf{lits}(F)$ returns the set of literals that occur in the formula F. Similarly, $\mathsf{atoms}(F)$ returns the set of variables that occur in the formula F.

An *interpretation* J is a (partial or total) mapping from the set of variables to the set $\{\top, \bot\}$ of truth values; the interpretation is represented by a sequence of literals, also denoted by J, with the understanding that a variable x is mapped to $\top$ if $x \in J$ and is mapped to $\bot$ if $\overline{x} \in J$. If a variable x is neither mapped to $\top$ nor to $\bot$ by J, we say the variable is *undefined*. This notion is also lifted to literals. One should observe that $\{x, \overline{x}\} \not\subseteq J$ for any x and J.

A clause C is *satisfied* by an interpretation J if $l \in J$ for some literal $l \in C$. An interpretation *satisfies* a formula F, if it satisfies every clause in F. If there exists an interpretation that satisfies F, then F is said to be *satisfiable*, otherwise it is said to be *unsatisfiable*. An interpretation J that satisfies a formula F is called *model* of F. We also say J models F and write $J \models F$. Given two formulae F and G, such that all models of F are also models for G, then, we say that the formula F models G and write $F \models G$. Two formulae F and G are *equivalent*, if they have the same set of models. This relation is denoted by $F \equiv G$. Assume, the formula F models the formula G: $F \models G$. By adding G to F, the resulting formula is equivalent to F: $F \cup G \equiv F$. Thus, adding a formula G that is modeled by F to F does not change the set of models for the formula F. The *reduct* $F|_J$ of a formula F with respect to an interpretation J is the formula obtained from F by evaluating F under J and simplifying the formula as follows: all satisfied clauses are removed, and from all the remaining clauses all literals x with $\overline{x} \in J$ are removed. Let $C = [x, c_1, \ldots, c_m]$ and $D = [\overline{x}, d_1, \ldots, d_n]$ be two clauses. We call the clause $E = [c_1, \ldots, c_m, d_1, \ldots, d_n]$ the *resolvent* of C and D, which has

been produced by *resolution* on variable x. We write $E = C \otimes D$. Note, that $\langle C, D\rangle \models \langle E\rangle$, and therefore $\langle C, D\rangle \equiv \langle C, D, E\rangle$.

2.2 Solving the SAT Problem

Satisfiability testing tries to answer the question whether there exists an interpretation J that satisfies a given formula F. For a formula F with n propositional variables this problem can also be stated as a search problem, where the search space is the set of all possible interpretations and the solutions space is the set of all models. The search space contains 2^n solution candidates. Even if the number of variables in formulae from applications raises up to half a million, modern SAT solvers usually can solve them in reasonable time.

Instead of working with total interpretations, where all variables of a formula F are defined, structured SAT solvers create a partial interpretation based on the *Davis-Putnam-Loveland-Logemann (DPLL)* algorithm [6]. This process can also be understood as creating a binary search tree and traversing it in depth-first manner. Regarding the whole search tree, each of its branches represents a (partial) interpretation. Let F be the given formula and J the interpretation represented by a branch B. We distinguish the following cases: (i) If J evaluates F to $\top$, then a model has been found and F is satisfiable. (ii) If J does not map F to a truth value (and no clause in F is mapped to $\bot$), then B is expanded by the so-called *decide rule*: a currently unassigned variable is assigned a new truth value and a backtrack point is recorded. Afterwards, J is extended by all the implications that can be found with respect to the formula. Extending the interpretation is mainly done by the *unit propagation rule*. (iii) If a clause of F is mapped to $\bot$ by J, then this clause is called *conflict (clause)* and B can be closed. Thereafter, naive backtracking is applied to explore the most recent alternative branches in the search tree.

2.3 Learning Information during Search

The idea to further analyze the conflict clause led to the *conflict driven clause learning (CDCL)* algorithm that was first presented in the SAT-solver GRASP [8]. By applying resolution to the conflict clause and to the clauses which have been used in the implications, new clauses are *learned*. Adding these *learnt clauses* to the formula leads to an improved backtracking behavior, where many branches of the search tree are closed by a single conflict.

Describing the CDCL algorithm in full detail is beyond the scope of this paper. However, four rules to traverse the search tree given in Table 1 are necessary for the rest of the paper. Solver implementations schedule these rules in a predefined order. A state in this rule system is defined as a pair of a formula F and an interpretation J, which represents the current branch in the search tree: $F :: J$. The rules can be understood as transition rules from the current state $F :: J$ to a successor state $F' :: J'$, which triggers under certain conditions. The full algorithm description based on these rules can be found in [26].

Table 1. Abstract Reduction System for CDCL algorithm

(1) $F :: J$	$\rightsquigarrow_{dec}$	F	$:: J\,\dot{l}$	iff $l \in \mathsf{atoms}(F) \cup \overline{\mathsf{atoms}(F)}$ and $l \notin J$ and $\bar{l} \notin J$
(2) $F :: J$	$\rightsquigarrow_{unit}$	F	$:: J\,l$	iff $[l] \in F\|_J$.
(3) $F :: J$	$\rightsquigarrow_{learn}$	$F, C :: J$		iff $F \models C$ and $C \subseteq \mathsf{lits}(F)$.
(4) $F :: J'\,\dot{l}\,J$	$\rightsquigarrow_{back}$	F	$:: J'\,l'$	iff $[l'] \in F\|_{J'}$.

The first rule $\rightsquigarrow_{dec}$ guides the search by creating a new branch on the current path in the search tree and assigns an undefined literal l of the formula F. Decision literals l are labeled with a dot: $\dot{l}$. The next rule $\rightsquigarrow_{unit}$ performs deduction, and at the same time prunes the search tree, because the only way to satisfy a clause $[l] \in F|_J$ is by extending J as $J \cup l$. The clause C, whose reduct $C|_J = [l]$ led to this *unit propagation*, is called the *reason (clause)* of the literal l. In case the algorithm hits a branch that contains a conflict, a new clause is learned with $\rightsquigarrow_{learn}$. This clause C is obtained by resolving the current conflict clause with the reasons of the literal assignments that led to the conflict. The rule $\rightsquigarrow_{back}$ can be applied to escape from the current part of the search tree.

Besides clause learning, other improvements have been added to the CDCL algorithm to achieve a better performance. These techniques include restarts [27], advanced branching heuristics [28] and simplifications during search [29]. Katebi et.al. show in [9] that among all the major improvements to SAT solvers, clause learning is the most beneficial technique. Furthermore, it has been shown in [30,31] that CDCL solvers can answer the SAT question with a lower complexity than the DPLL algorithm. These two results demonstrate that learning new information is beneficial for the search process of SAT solvers.

2.4 Parallel SAT Solving

With the availability of parallel hardware, parallel SAT solvers have been invented, starting in 1994 [32]. An overview of parallel SAT solving since that time is given in [33,34]. Parallelizing the search process inside the DPLL algorithm has been done in [16], however this approach does not scale beyond two cores. Since modern hardware provides many more cores, we focus on techniques that are more promising, namely:

- **parallel portfolio search** [13], where different solvers solve the same input formula in parallel
- **plain partitioning** [21], where the input formula is partitioned into subformulae and afterwards each sub-formula is solved by a solver
- **iterative partitioning** [21], where a formula is partitioned iteratively into a tree of sub-problems and every sub-problem is solved in parallel.

Portfolio parallelization is the most common approach and many parallel SAT solvers rely on this technique, e.g. [13,35]. Plain partitioning is a basic partitioning approach: The formula F is divided into n sub-problems $F_1, \ldots, F_n$ where the

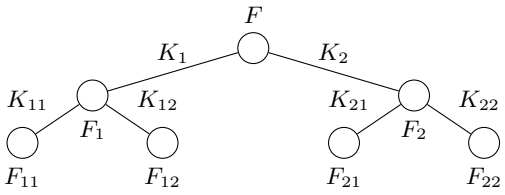

Fig. 1. The tree shows how a formula can be partitioned iteratively by using a partitioning function that creates two child formulae.

following constraint has to be met: $F \equiv F_1 \vee \cdots \vee F_n$. Usually, the search spaces of sub-formulae are disjunct: i.e. $F_i \wedge F_j \equiv \bot$, where $1 \leq i < j \leq n$. Hyvärinen et.al. show that plain partitioning suffers from a theoretical slow down [18]. The third parallel solving approach, iterative partitioning, solves a given formula and creates sub-problems that are solved in parallel as well. Iterative partitioning does not have the theoretical slow down of plain partitioning. Furthermore, this approach seems to scale better than plain partitioning or the portfolio search, if the number of available cores increases [18]. A more formal description of the iterative partition approach is given in the next section.

3 Sharing Information in Parallel Search

The partitioning of the search space of a formula F is illustrated by the *partition tree* in Figure 1. A *partition function* splits a formula F into n sub-problems $F_1, \ldots, F_n$ meeting the following constraints: $F \equiv F_1 \vee \cdots \vee F_n$ and $F_i \wedge F_j \equiv \bot$, for each $1 \leq i < j \leq n$. W.l.o.g one can assume that every partition F_i is of the form $F \wedge K_i$, for some *cnf constraint* K_i. 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 $\phi(F')$. A more convenient notation for nodes in a tree is given by marking them with their *positions*: the root node has the empty position ϵ, whereas the node at position pi is the *i-th* successor of the node at position p. The set of all positions in $\mathcal{T}$ is $pos(\mathcal{T})$. With F^p we denote the node at position p of a tree rooted in F. Observe that, for every position $p \in pos(\mathcal{T})$, 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 $p := i_1 ... i_n$ and each $i_j \in \{1, \ldots, |\phi(F^{i_1 \ldots i_{j-1}})|\}$. Since a partition tree is created upon a partition function, clearly $F^p \equiv \bigvee_i F^{pi}$ and $\forall_{i \neq j} F^{pi} \wedge F^{pj} \equiv \bot$, for every $p \in pos(\mathcal{T})$, $i, j \in \{1, \ldots, |\phi(F^p)|\}$. Sharing learned clauses among solvers that solve child formulae has been considered briefly in [22]. There, Hyvärinen et.al. introduce an expensive mechanism called *assumption-based (learned) clause tagging* and a fast approximation method *flag-based (learned) clause tagging*.

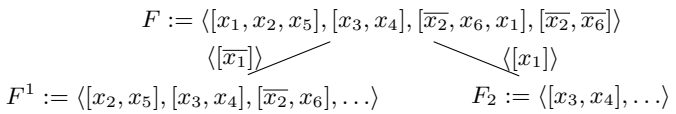

Fig. 2. Partition tree for F. The successor F^{pi} of a node F^p is created by applying resolution on each clause in F^p with each unit clause in partition constraint K^{pi}.

3.1 Flag-Based Clause Tagging

Consider the formula $F^1 = \langle [x_2, x_5], [x_3, x_4], [\overline{x_2}, x_6], [\overline{x_2}, \overline{x_6}] \rangle$ in the partition tree of Figure 3.1 and the following local sequential run:

$$F^1 :: (\ ) \rightsquigarrow_{dec} F^1 :: (\dot{\overline{x_5}}) \rightsquigarrow_{unit} F^1 :: (\dot{\overline{x_5}}, x_2) \rightsquigarrow_{unit} F^1 :: (\dot{\overline{x_5}}, x_2, x_6)$$

Observe, this run leads to a conflict after the decision $\dot{x_5}$ and unit propagations x_2 and x_6 so that the clause $[\overline{x_2}] := [\overline{x_2}, \overline{x_6}] \otimes [\overline{x_2}, x_6]$ is learned. Since $F \not\models [\overline{x_2}]$, this clause cannot be added to the clauses of F. This example motivates related work [36]: If the clause to be shared does not depend on a partition constraint the problem can be avoided. To keep track of these clauses, Boolean flags have been introduced in [36], which indicate whether a clause can be shared "*safely*". This approach is called *flag-based tagging*.

Definition 1 (Unsafe clauses). *Consider a node F^p of a partition tree rooted in F. Then a clause $C \in F^p$ is **unsafe** if and only if:*

1. *C belongs to a partition constraint,*
2. *C is a learned clause obtained as the result of a resolution derivation involving unsafe clauses.*

*A clause that is not unsafe is called **safe**.*

If a clause C is safe, then for every position p we have that $F^p \models C$. Figure 3 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 $[x_4, x_2, \overline{x_5}]$:

$$F^{21} :: () \rightsquigarrow_{unit} F^{21} :: (\overline{x_2}) \rightsquigarrow_{dec} F^{21} :: (\overline{x_2}, \dot{\overline{x_4}}) \rightsquigarrow_{unit} F^{21} :: (\overline{x_2}, \overline{x_4}, x_5)$$

The learnt clause is $D = [x_4, x_2] = [x_4, x_2, x_5] \otimes [x_4, x_2, \overline{x_5}]$. Since only safe clauses have been used in the resolution, D is a safe clause and thus it can be shared among every node in the partition tree. Observe that clause $[x_4, x_2]$ speeds-up the computation on node F^1. Consider Figure 3, and the following sequential execution over node F^1 after incorporating the shared clause $[x_4, x_2]$:

$$F^1 :: (\ ) \rightsquigarrow_{dec} F^1 :: (\dot{\overline{x_4}}) \rightsquigarrow_{unit}$$

$$F^1 :: (\dot{\overline{x_4}}, \overline{x_2}) \rightsquigarrow_{back} F^1 :: (\ ) \rightsquigarrow_{learn} F^1, [x_4] :: (\ )$$

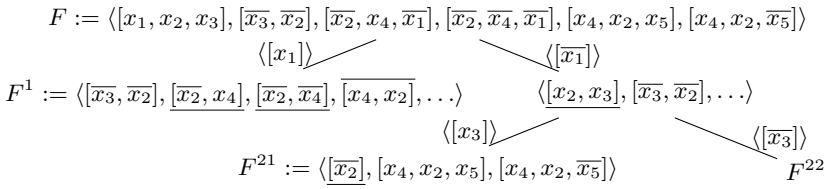

Fig. 3. Partition tree over F with clause-tagging. Unsafe clauses are underlined. The overlined clause $\overline{[x_4, x_2]} \in F^1$ is a shared clause that has been incorporated from F^{21}.

After the decision $\dot{\overline{x_4}}$, the local solver can immediately use the shared clause $[x_4, x_2]$ to derive the learnt clause $[x_4]$. Performing the same decisions and propagating without using the safe shared clause would lead to the learnt clause $[x_4, x_2]$. Hence, flag-based clause sharing can effectively speed-up the local computation of some node in the tree.

A weakness of the flag-based tagging is shown in Figure 4, where we slightly changed the shape of the partition tree. Assume the clause $D = [x_4, x_2]$ is learnt while working on formula F^{121}. Since the resolution $\underline{[x_4, x_2, x_5]} \otimes [x_4, x_2, \overline{x_5}]$ involves an unsafe clause, D is also tagged as unsafe and thus it is not shared at all. However, from previous examples we know that this clause can be "safely" shared among all the formulae 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 following situations cannot be covered:

1. An unsafe clause can be a semantic consequence of the original formula, and thus be shareable
2. An unsafe clause is not shared at all. However, it might be considered safe for some sub-tree of the original partition tree, and thus be shared among the nodes belonging to this sub-tree.

The first problem can only be solved by an algorithm which is more complex than the presented approximation. As shown in [22], using the approximation instead of the complex mechanism still results in higher performance, because the benefits of the complete algorithm cannot overcome its overhead. Solving the second problem can be done by extending the tagging, which we do in the next section.

3.2 Position-Based Clause Tagging

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

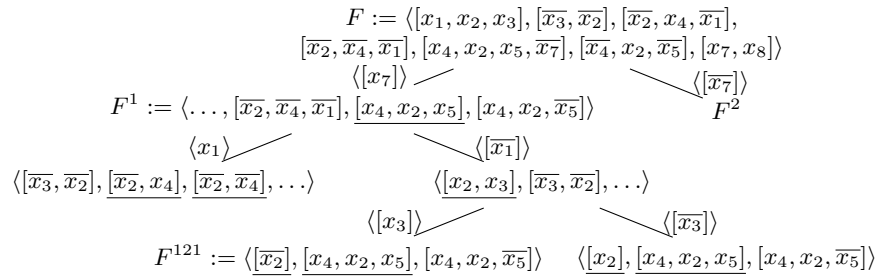

Fig. 4. Clause $[x_4, x_2]$, learnt by the local solver working on the node solving formula F^{121}, is not safe anymore, because it depends on the partition constraint x_7.

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 F, clauses belonging to 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. In general, a clause C^p will only be added to a formula F^q if $F^q \models C^p$ (that is, the set of models of F does not change by adding C^p). The *receive rule* that is used for position-based tagging is the following:

Definition 2 (Receive Rule). *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*

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

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 7 below. In order to prove it, we make use of an auxiliary definition:

Definition 3 (Resolution Order). *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 [37] is valid on $>_{res}^{+}$.

Lemma 4. *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 $(R_1^{r_1}, \ldots R_j^p, \ldots, R_n^{r_n})$, 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 5. *If C^p is a learnt clause that has been obtained from a resolution derivation $(R_1^{p_1}, \ldots, R_n^{p_n})$, then p_i is a prefix of p, for every $1 \leq i \leq n$.*

Proof. This is a consequence of Lemma 4

Theorem 6. *Given a clause C^p and a node F^p, 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 $(R_1^{q_1}, \ldots, R_n^{q_n})$, and that the theorem holds for each of these resolvents. From Lemma 5, we have that $q_1, \ldots, q_n$ 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 7. *Given a clause C^p and a formula F^{pq}, it holds $F^{pq} \models C^p$.*

Proof. It follows directly from Theorem 6 and from the equality $F^{pq} = F^p \cup K^{pq}$.

Now reconsider the example in Figure 4, which is an extension of Figure 3. Flag-based clause tagging was not able to share the learnt clause $[4, 2]$ anymore, because $[4, 2]$ is unsafe. The new sharing rule with position-based tagging can share this clause again as in the situation of Figure 3: all solvers working on formulae F^{1p} can receive this clause.

3.3 Implementation Details

For flag-based tagging only a single Boolean program variable is used to store whether a learnt clause is safe. In theory, position-based tagging tags each clause with a position and does expensive position operations during conflict analysis (for assigning the right position) and during the receive rule application (only those clauses tagged with a position prefix of the current position are accepted).

The implementation of this approach is less complicated and has no overhead compared to the flag-based approach: each node in the partition tree provides a clause *storage*, where all shared clauses that are tagged with the position of this node are stored. Instead of encoding positions, it is sufficient to tag clauses with an integer storing the position length (i.e., a level in the partition tree): a clause tagged with an integer n has to be sent to the storage of the ancestor of level n of the current node in the partition tree. When a solver incorporates shared clauses, it only receives clauses from storages that belong to the position from the current node to the root of the partition tree.

Again, from incorporated clauses only the length of the position is sufficient to tag learnt clauses correctly. Instead of considering the maximum position, only the maximum length has to be selected, which is a simple integer comparison and thus not more expensive than comparing Boolean variables.

All the storages do not store all shared clauses over the whole run. Ring-buffers with a size of 10000 are used, so that the first clause is overwritten with the 10001^{th} clause. Local solvers often incorporate clauses from a storage, but seldom add clauses to the pools so that reader-writer locks protect the pools instead of usual mutexes. Experiments showed that reader-writer locks give an improvement of up to 10 % against mutual exclusion semaphores.

4 Empirical Evaluation

The experiments have been run in a multicore 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 solvers. Every approach has been tested over 600 instances, that is the whole set of instances of SAT challenge [24]. 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 case execution times have been quite stable, and thus the results here exposed are likely to be replicated.

We implemented a parallel SAT solver, based on the work in [18] that is based on Minisat [15]. That solver already shares learnt unit clauses downwards in the partition tree, after a solver finished processing a node [18]. The evaluation includes three further solver configurations:

Table 2. Number of solved instances

Approach	Solved	SAT	UNSAT	Average run time	CPU ratio	Score
POS	**430**	239	**191**	377.397	11.5	**78**
RAND	380	232	148	**374.445**	11.5	-50
FLAG	417	234	183	378.969	11.4	30
NONE	418	**244**	174	383.785	**12.1**	-58

1. POS, where the presented position-tagging is used
2. RAND, where any learnt clause is shared position-based with 5% probability
3. FLAG, where we use the sharing approach of [22]
4. NONE, where no clauses are shared

Note, local solvers will only share clauses with two or less literal. Only RAND shares clauses of any size. Clauses are put into the clause storage as soon as they have been learnt. A nice feature of position based tagging is that it allows a certain degree of flexibility. Indeed, since $F^p \models C^p$, the clause C could be put in any storage at position pp' (provided that pp' is a valid position in the partition tree) without affecting soundness. In our experiments we make use of this flexibility by worsening the sharing level for POS and RAND: if a clause C should be sent to level k, then we send it to level $k' = k + log_2|C|$. We do this in order to fill the various pools in a more homogeneous way. As in [18] the resources of the local solvers are restricted: a branch is created after 8096 conflicts, and a local solver is allowed to search until 512000 conflicts have been reached.

Table 2 gives various properties of the four configurations on the benchmark. For the whole benchmark, as well as separately for satisfiable (SAT) and unsatisfiable (UNSAT) formulae the number of solved instances is given. POS slightly outperforms every other approach by solving at least 12 instances more. Surprisingly, this ranking gives a poor performance to previous work FLAG [22]. 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 sharing heuristic is also important, as can be seen when POS is compared to RAND: POS solves more instances and the average run time per instance is almost the same. Another interesting measure for parallel solvers is its scalability. The *CPU ratio* shows how many cores have been used in average to solve all the instances. 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 [25]. A more careful ranking, which also takes solving times into account, gives a different picture. The used noise value for ties has been set to 60 seconds, so that instances that are solved faster than this value are not considered. Now, FLAG shows the second best performance after POS, which looks more like the expected evaluation. Furthermore, this ranking shows that the new approach outperforms the other configurations significantly. Comparing POS directly with NONE the score is 37 to -37 points. Against FLAG, POS still wins with 13 to -13 points. The improvement in the run time of the different sharing approaches is furthermore compared in the cactus plot in Figure 5. Each dot (x, y) in the diagram shows that a configuration solves x instances with a timeout of y seconds for each instance. This plot shows also for other timeouts that POS is the superior configuration. A reason for the improvement of the search is that clauses are shared. We analyzed how many clauses have been shared, and furthermore recorded the subtree where these clauses are valid. Table 3 shows the average number of shared clauses per run of

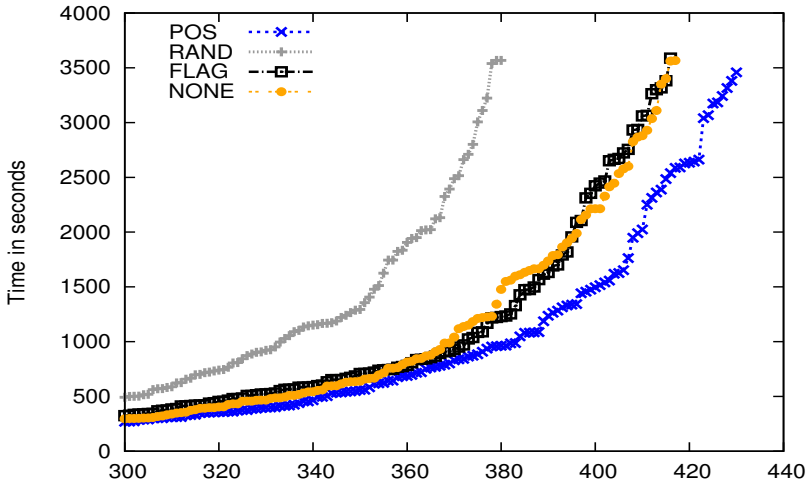

Fig. 5. Solved instances and solving times of the four configurations

a configuration. 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 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 storages of the first child nodes. All remaining clauses are sent at a higher level in the tree and thus shared among fewer nodes. Restricting shared clauses to binary and unary clauses in POS 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 POS and RAND share binary clauses not at the root node, so that the number cannot be easily compared to FLAG. Still, POS shares more clauses than FLAG in total, and also results in a higher performance. Summarizing, it is shown that our new sharing approach can share more clauses than previous approaches. However, simply sending any clause without a good filter results in a degradation of performance. The used size restriction seems to be a good filter heuristic.

Table 3. Average number of shared clauses

Configuration	shared level 0	shared level 1	total shared
POS	1420	5663	17202
RAND	29472	31676	209199
FLAG	6557	0	6557
NONE	0	0	0

Table 4. Scalability of the parallel solver

Configuration	SAT slower	SAT faster	UNSAT slower	UNSAT faster	SAT run time 4-core	SAT run time 16-core	UNSAT run time 4-core	UNSAT run time 16-core
POS	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

4.1 Scalable Search

To check whether for future multi-core architectures the approach will scale further, we run all four solver configurations also with a restriction to 4 cores and measured 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 4. Furthermore, the average run time is compared. The data shows that most of the instances benefit from additional resources. The run time comparison shows that most of the time 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. Since the search of local solvers is not structured, but enhanced with clause learning and improved backtracking, linear speedups cannot be expected. These results are in line with the results of [18], where no clause sharing was used.

5 Conclusion

We presented a new position-based clause sharing technique that allows to share clauses for subsets of a parallel search space partitioning SAT solver. Position-based clause sharing improves the intensification of parallel searching SAT solvers by identifying the search space in which a shared clause is valid so that the total number of shared clauses can be increased compared to previous work [22]. Experiments with parallel SAT solvers show that clause sharing is important for the performance of these solvers. Thus, position-based clause sharing is a nice extension of clause sharing, which could also be incorporated into other search space partitioning search procedures.

Future work could improve shared clauses further. By rejecting resolution steps, the sharing position of learnt clauses can be improved. Additionally, parallel resources should be exploited further, for example by using different partitioning strategies or by replacing the local sequential solver by another parallel SAT solver. Furthermore, more sophisticated search space partitionings have to be analyzed and evaluated. Finally, improvements to the local solver, as for

example restarts and advanced search direction techniques, could also be incorporated into the search space partitioning.

References

1. Großmann, P., Hölldobler, S., Manthey, N., Nachtigall, K., Opitz, J., Steinke, P.: Solving periodic event scheduling problems with sat. In Jiang, H., Ding, W., Ali, M., Wu, X., eds.: IEA/AIE. Volume 7345 of Lecture Notes in Computer Science., Springer (2012) 166–175
2. Goel, A.: A column generation heuristic for the general vehicle routing problem. In Blum, C., Battiti, R., eds.: LION. Volume 6073 of Lecture Notes in Computer Science., Springer (2010) 1–9
3. Carlier, J., Pinson, E.: An algorithm for solving the job-shop problem. Manage. Sci. **35**(2) (1989) 164–176
4. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications., IOS Press (2009)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
7. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. [5] chapter 4 131–153
8. Marques Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5) (1999) 506–521
9. Katebi, H., Sakallah, K.A., Marques-Silva, J.a.P.: 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, Berlin, Heidelberg, Springer-Verlag (2011) 343–356
10. Li, C.M., Manyà, F.: Maxsat, hard and soft constraints. [5] chapter 19 613–631
11. Kuegel, A.: Improved exact solver for the weighted max-sat problem. In Berre, D.L., ed.: POS-10. Volume 8 of EPiC Series., EasyChair (2012) 15–27
12. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT **7**(2-3) (2010) 59–6
13. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. JSAT **6**(4) (2009) 245–262
14. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical Report 10/1, Johannes Kepler University, Linz, Austria (2010)
15. Eén, N., Sörensson, N.: An extensible sat-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
16. Manthey, N.: Parallel SAT Solving - Using More Cores. In: Pragmatics of SAT(POS'11). (2011)
17. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. AI **45**(3) (1990) 275–286
18. Hyvärinen, A.E.J., Manthey, N.: 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. (2012) 214–227

19. Chrabakh, W., Wolski, R.: 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, ACM (2003) 37–
20. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel sat solving. In Cohen, D., ed.: CP. Volume 6308 of Lecture Notes in Computer Science., Springer (2010) 252–265
21. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning sat instances for distributed solving. In Fermüller, C.G., Voronkov, A., eds.: LPAR (Yogyakarta). Volume 6397 of Lecture Notes in Computer Science., Springer (2010) 372–386
22. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Grid-based sat solving with iterative partitioning and clause learning. In Lee, J.H.M., ed.: CP. Volume 6876 of Lecture Notes in Computer Science., Springer (2011) 385–399
23. Arbelaez, A., Hamadi, Y.: Improving parallel local search for sat. In Coello, C.A.C., ed.: LION. Volume 6683 of Lecture Notes in Computer Science., Springer (2011) 46–60
24. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012) 89–92
25. Van Gelder, A.: Careful ranking of multiple solvers with timeouts and ties. In: Proceedings of the 14th international conference on Theory and application of satisfiability testing. SAT'11, Berlin, Heidelberg, Springer-Verlag (2011) 317–328
26. Arnold, H.: A linearized dpll calculus with clause learning. (2010)
27. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1-2) (2000) 67–100
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. (2001) 530–535
29. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In Gramlich, B., Miller, D., Sattler, U., eds.: Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012). Volume 7364 of Lecture Notes in Computer Science., Springer (2012) 355–370
30. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning sat solvers as resolution engines. Artif. Intell. **175**(2) (2011) 512–525
31. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Int. Res. **22**(1) (2004) 319–351
32. Bhm, M., Bohm, M., Speckenmeyer, E., Speckenmeyer, E.: A fast parallel sat-solver - efficient workload balancing (1994)
33. Martins, R., Manquinho, V., Lynce, I.: An overview of parallel sat solving. Constraints **17** (2012) 304–347
34. Hölldobler, S., Manthey, N., Nguyen, V., Stecklina, J., Steinke, P.: A short overview on modern parallel SAT-solvers. In et.al., I.W., ed.: Proceedings of the International Conference on Advanced Computer Science and Information Systems. (2011) 201–206 ISBN 978-979-1421-11-9.
35. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.M., Piette, C.: Revisiting clause exchange in parallel sat solving. In: Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing. SAT'12, Berlin, Heidelberg, Springer-Verlag (2012) 200–213
36. Hyvärinen, A.E., Junttila, T., Niemelä, I.: Incorporating learning in grid-based randomized sat solving. In: Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications. AIMSA '08, Berlin, Heidelberg, Springer-Verlag (2008) 247–261

37. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York, NY, USA (1998)