Master Thesis

State-of-the-art

Cooperative Parallel SAT Solving

Ahmed Irfan

03 April 2013

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

Supervised by:
Prof. Dr. rer. nat. Steffen Hölldobler
Dipl.-Inf. Norbert Manthey

Ahmed Irfan
State-of-the-art Cooperative Parallel SAT Solving
Master Thesis, Fakultät für Informatik
Technische Universität Dresden, April 2013

Task of the Master Thesis

Surname, Name:	Irfan, Ahmed
Course of Studies:	International Masters in Computational Logic
Matriculation #:	3732599
Title:	***State-of-the-art Cooperative Parallel SAT Solving***
Task Description:	The question whether a propositional formula in conjunctive normal form is satisfiable (SAT) is answered with powerful clause learning SAT solvers. Over the years, many improvements, including preprocessing, restarts, advanced decision heuristics and clause management, have been added to these systems. There also exist parallel solvers that can be categorized into cooperative and competitive approaches. Competitive solvers have been researched for a long time and have been improved by clause sharing heuristics and other information sharing. However, the cooperative approach seems to scale better when more parallel resources become available. In this thesis the current state of the art for cooperative parallel SAT solvers should be improved by trying to integrate improvements from sequential and portfolio solvers into a search space partitioning solver. Finally, an empirical evaluation should show whether the cooperative approach is competitive and whether it indeed scales better than the competitive approach.

Abstract. Parallel SAT solvers can be categorized into cooperative and competitive approaches. Over the past few years, competitive SAT solvers have been researched and improved by clause sharing heuristics and other information sharing. In the meanwhile, cooperative approach had been receiving a little attention. This work attempts to revisit cooperative approach using a not-so-new promising technique called iterative partitioning [HJN11] and a novel clause sharing technique [LM13]. By improving iterative partitioning, we have shown that cooperative approach is comparable, in terms of performance, to state-of-the-art competitive SAT solvers. Our solver SplitterGluLA based on cooperative approach, on a benchmark of 880 instances and with a time limit of 7200 seconds, solves 10 additional instances than Plingeling (a competitive approach based solver and winner of *SAT competition 2011*-parallel track), but solves 22 less instances than PeneLoPe (another competitive approach based solver and runner-up of *SAT challenge 2012*-parallel track). We have also shown that SplitterGluLA scales (when more resources are added) slightly better than PeneLoPe (currently best known scalable competitive SAT solver).

Contents

1. Introduction

The propositional satisfiability problem (often abbreviated as the SAT problem) is one of the most researched NP-complete problem in theoretical computer science. It is a problem of deciding if there exists a truth assignment under which a propositional formula evaluates to true. The SAT problem is of great importance, as it is the best known example of NP-complete problem [Coo71] and it is considered to be the key for the still open *P vs NP* problem (Millennium Prize Problem) from Clay Mathematics Institute. The SAT problem in principle is hard to solve, yet modern state-of-the-art *SAT solvers* (procedures that solve the SAT problem) solve real life instances quite efficiently. For example, SAT solvers are used in planning [KS92], scheduling [GHM+12] [CP89], vehicle routing [Goe10], hardware and software verification [BCCZ99] [DKW08], bioinformatics [LMS06] and configuration [ABL+10]. SAT solvers are used as a black box by encoding the problem from application domain to the SAT problem. This paradigm of encoding a problem to the SAT problem is often called declarative problem solving. This paradigm relieves a programmer from the algorithm design, as this part is done by SAT solvers, and the programmer has to work on encoding. Figure 1 shows an example of the usage of SAT solvers. Any given problem from application domain like planning or bioinformatics, is encoded (shown by Encoder in the figure) into the SAT problem which is then given to a SAT solver. It is job of the SAT solver to give the solution, which is translated back from the SAT problem to application domain (shown by Decoder in the figure).

SAT solvers can be categorized in two main types: *stochastic* and *systematic*. SAT solvers in the stochastic type are based on the local search algorithms [SK93] which are incomplete algorithm, e.g. random walk; while SAT solvers in systematic type are based on backtracking search algorithm like Davis Putnam Logemann Loveland (DPLL) algorithm [DP60] [DLL62], which is a complete algorithm. For a given instance of the SAT problem, complete SAT solvers can return *yes* if there exists a satisfying variable assignment, or return *no* if no such assignment exists.

Over the last two decades, a lot of improvements have been made to SAT solvers and can be seen in the yearly SAT competitions [sat]. This success is more evident in the sequential SAT solvers (using one CPU) and quite less in parallel ones (using more than one CPU). With the technological shift from single core CPU to muti-core CPUs, parallel SAT solvers require more attention than before. This work focuses on developing a (systematic) complete parallel SAT solver.

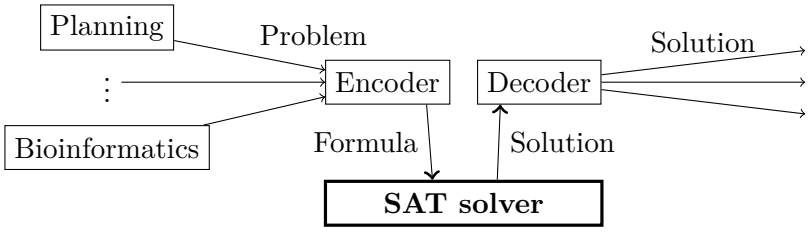


Figure 1: Example - Applications of SAT solver

1.1. Related Work

The first successful attempt to build a complete parallel SAT solver was based on the concept of dividing the search space of a complete SAT solver into sub-search spaces and solve each sub-search space in parallel [ZBP+96]. This approach is called *search space partitioning* and more specifically *plain partitioning* [HJN11]. This was the most prominent approach until the year 2008, when ManySAT [HJS08] emerged as the winner of parallel track of *SAT race 2008*. ManySAT uses a different approach; it runs several different SAT solvers in parallel for the same problem and wait for the first answer. This approach is called *portfolio* approach. We categorize search space partitioning approach as *cooperative* and portfolio approach as *competitive*, for parallel SAT solving. Since the success of ManySAT, most of the research in parallel SAT solvers has been focused on portfolio approach. This is evident from the solvers participating in the yearly SAT competitions, as the number of parallel SAT solvers based on portfolio approach has increased while other parallel SAT solvers has decreased significantly, e.g. in *SAT challenge 2012*, 17 out of 19 parallel SAT solvers were based on portfolio approach.

The reason for slower performance of search space partitioning approach w.r.t. portfolio approach has been given in [HJN09], which proves that search space partitioning suffers from a theoretical slowdown. Later the author proposed a new search space partitioning approach called *iterative partitioning* [HJN11], which does not suffer from slowdown. The idea of iterative partitioning is similar to plain partitioning as both divide the search space into sub-search spaces, but the former solves the original search space with the sub-search spaces in parallel, but the latter only solves the sub-search spaces in parallel. Based on the idea of iterative search space partitioning, an effort has been made to build a parallel SAT solver Splitter [HM12a] [HM12b]. Splitter performed poorly in *SAT challenge 2012* and was placed at the last position in the parallel track.

1.2. Contributions

Although iterative partitioning have strong theoretical results [HM12a], but Splitter (parallel solver based on iterative partitioning) did not perform up to expectation in *SAT challenge 2012*. This work tries to find reasons for the difference in theoretical vs practical results. The goal of this work is:

- to develop a complete parallel SAT solver based on iterative partitioning approach that is competitive with the parallel SAT solvers based on portfolio approach.
- to apply the recent techniques used in sequential SAT solver to the parallel SAT solver based on iterative partitioning approach.
- to explore the concept diversification and intensification, used by portfolio based SAT solvers, in the parallel SAT solvers based on iterative partitioning.
- to check if the solvers based on iterative partitioning approach scale better than the solver based on portfolio approach.

1.3. Structure of this report

The structure of this report is as follows: we will start with the basic background knowledge about propositional logic and SAT solving in Section 2. In Section 3, we will discuss the recent improvements in SAT solving, particularly in sequential SAT solving. We will discuss in detail the concepts in parallel SAT solving, in Section 4. Section 5 will focus on the contributions of this work. In Section 6, we provide a detailed evaluation of iterative partitioning based parallel SAT, i.e. performance, scalability, and comparison with portfolio based parallel SAT solvers. In the end, we give conclusion and future direction of this work in Section 7.

2. Preliminaries

In this section, we provide the necessary background knowledge and notations used in this report so that you find the report easy to understand. Whenever we introduce a new concept, we either provide its definition with example or where we can not provide definition or example, we give you references for details.

In the section 2.1, we introduce the notions of propositional logic. As most SAT solvers take *formulas* only in *conjunctive normal form (CNF)*, we restrict the description to CNF. For further reading, you can refer to the chapter 2 of the *Handbook of Practical Logic and Automated Reasoning* [Har09]. We discuss the basic Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving the SAT problem in Section 2.2.

2.1. Propositional Logic

We start with some basic definitions. You should note that some of the notions introduced here are more specific than the notions found in literature.

Definition 2.1. The set of **atomic propositions** AP is a countably infinite set of symbols.

We also call atomic propositions as *atoms* or *propositional variables* or simply *variables*. Propositional variables can be assigned the truth value either *true* or *false*, we represent true by $\top$ and false by $\bot$. We have unary opertion *negation* on AP and the operation is represented by $\neg$. We also have binary operations *disjunction* and *conjunction* on AP and these operations are represented by $\vee$ and $\wedge$, respectively.

Definition 2.2. A **literal** L is an atomic proposition A or its negation $\neg A$.

Definition 2.3. A **clause** is a disjunction of literals.

Definition 2.4. A **CNF formula** F is a conjunction of clauses.

We represent a clause by a finite set of literals, and CNF formula by a set of clauses. A clause is called a valid clause: if it contains a complementary pair of literals. *Size* of a clause is defined as the number of literals in the clause, and we denote the size of clause C by $|C|$. A clause C is called a *unit clause*, or *binary clause*, or *ternary clause*: if and only if $|C| = 1$, or $|C| = 2$, or $|C| = 3$, respectively.

Example 2.5. Consider the following CNF formula:

$$(1 \vee \neg 2) \wedge (3) \wedge (4 \vee \neg 5 \vee \neg 7)$$

we represent the given CNF formula in the set notation, presented in this section; let F be the CNF formula:

$$F = \{\{1, \neg 2\}, \{3\}, \{4, \neg 5, \neg 7\}\}$$

you should note that the clauses are represented as a set of literals and the CNF formula is represented as set of set of literals; also note that $\{3\}$ a unit clause, and $\{1, \neg 2\}$ is a binary clause, and $\{4, \neg 5, \neg 7\}$ is a ternary clause.

We call the *polarity* of a literal L to be *positive* if $L = A$, or *negative* if $L = \neg A$, where $A \in AP$. In order to change the polarity of a literal, we define a function, complement of a literal. Given a literal L, we define *complement of literal*, represented as $\overline{L}$, as:

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

We say that a pair of literals is a *complementary pair of literals:* if, in the pair, the first literal is L and the second literal is $\overline{L}$.

Definition 2.6. An **interpretation** is a (partial or total) assignment of propositional variables to truth value.

We represent interpretation by a sequence of literals such that there are no duplicate literals and no complementary pair of literals in the sequence. Before introducing the notion of reduct, we first define the complement of a set of literals. The *complement of a set of literals* S, represented by $\overline{S}$, is defined as:

$$\overline{S} = \{\overline{L} \mid L \in S\}$$

Example 2.7. Consider the following set S, a set of literals: $S = \{4, \neg 5, \neg 7\}$ then the complement of S is the set: $\overline{S} = \{\neg 4, 5, 7\}$

Now, we define the notion of reduct. Given a clause C and an interpretation J, let $s(J)$ be the set of all the elements in J, then, we get the *reduct of clause* C w.r.t. J, denoted $C|_J$, by the following definition:

$$C|_J = \begin{cases} \top & \text{, if } C \cap s(J) \neq \emptyset \\ C \setminus \overline{s(J)} & \text{, otherwise} \end{cases}$$

we extend the notion of reduct for CNF formula. Given an interpretation J and a CNF formula F, the *reduct of CNF formula* $F|_J$ of F w.r.t. J is a CNF formula:

$$F|_J = \{C|_J \text{ , such that } C \in F \text{ and } C|_J \neq \top\}$$

We say an interpretation J *satisfies* a clause C, if and only if $C|_J = \top$. Let F be a CNF formula, then we call J a *model* for F, in symbols $J \models F$, if and only if $F|_J = \{\,\}$. Conversely, we say that J *falsifies* F, in symbols $J \not\models F$, if and only if $\{\,\} \in F|_J$. In the latter case, J is called *conflict* for F.

Lemma 2.8. *Let F be a CNF formula, and J an interpretation, then $J \models F$, if and only if J satisfies every clause in F.*

Example 2.9. We consider a CNF formula:

$$F = \{\{1, \neg 2\}, \{3\}, \{4, \neg 5\}\}$$

and an interpretation:

$$J = (1, \neg 3, 5)$$

then to obtain the reduct of F w.r.t. J, we first remove clauses from F: the clauses whose intersection with the set of J, $s(J) = \{1, \neg 3, 5\}$, is not an empty set; secondly, we remove the complement of the literals present in the set of J, $\overline{s(J)} = \{\neg 1, 3, \neg 5\}$, from the clauses in F. Thus, we obtain the reduct of F w.r.t. J:

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

You should note that the interpretation J falsifies F and is called conflict for F because there exists an empty clause (an empty set) in $F|_J$. In the same way, if we have an interpretation $J_1 = (1, 3, 4, 5)$, then J_1 satisfies F because $F|_{J_1}$ is an empty set. In this case J_1 is called model for F.

We have now every notion that is needed to define the satisfiability problem.

Definition 2.10. Given a CNF formula F, **the propositional satisfiability problem** or briefly **SAT** is the problem to decide whether F is *satisfiable*, i.e. if there exist an interpretation for F such that it satisfies F.

We say a CNF formula F is *unsatisfiable* if it is not satisfiable. We give now some other notions that will be used in this document. Let F, G be CNF formulas, then we say G is a *semantic consequence* of F, denoted as $F \models G$, if and only if for every interpretation J:

$$J \models F \text{ implies } J \models G$$

Let F, G be CNF formulas. Then F and G are *semantically equivalent*, in symbols $F \equiv G$, if and only if:

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

Let C_1 be the clause containing literal L and C_2 be the clause containing literal $\overline{L}$, then the *(propositional) resolvent* of C_1 and C_2 with respect to L is defined as:

$$\{C_1 \setminus \{L\}\} \cup \{C_2 \setminus \{\overline{L}\}\}$$

A clause C is said to be resolvent to C_1 and C_2, if and only if there exists a literal L such that C is the resolvent of C_1 and C_2 with respect to L.

2.2. SAT Solving

Solving the SAT problem is commonly termed as *SAT solving* and a computer program that solves the SAT problem is called *SAT solver*. The history of SAT solving goes way back to mid of the last century, but most noticeably is the work [DP60] and [DLL62],

where the latter is improved version of the first. The algorithm given in [DLL62] to solve the SAT problem is known as the DPLL algorithm and it is named after the authors Davis, Putnam, Logemann, and Loveland. Today, most of the modern SAT solvers are based on the DPLL algorithm.

We define a helping function *atom*, for describing DPLL algorithm. Given a literal L, *atom* returns its underlying propositional variable, formally we define it as:

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

We overload the function *atom* to handle clauses, i.e. given a clause C, *atom* will a return set of propositional variables:

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

Likewise, we also overload *atom* for CNF formula, i.e. given a CNF formula F then *atom* returns the set of atoms present in the formula:

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

We explain the DPLL algorithm with the help of abstract reduction system. Let R be a set and $\rightarrow$ be a binary relation $R \times R$, then the pair $(R, \rightarrow)$ is defined to be an *abstract reduction system* (ARS). The binary relation $\rightarrow$ is called reduction and every pair $(x, y) \in \rightarrow$ is written as $x \rightarrow y$. With $\rightarrow^+$, we represent the transitive closure of $\rightarrow$, and the reflexive and transitive closure of $\rightarrow$ is represented by $\rightarrow^*$. We suggest [BN98] for further reading about abstract reduction systems.

Definition 2.11. The **DPLL abstract reduction system** (DPLL ARS) is an ARS $(R_{DPLL}, \rightarrow_{DPLL})$, where:

- R_{DPLL} is a set:

$$R_{DPLL} \subseteq (CNFF \times PA) \cup \{\text{SAT}, \text{UNSAT}\}$$

such that $CNFF$ is a set of CNF formulas, PA is a set of interpretations over the

(1)	$F :: J$	$\rightsquigarrow_{\text{SAT}}$	SAT	iff	$F\|_J = \emptyset$
(2)	$F :: J$	$\rightsquigarrow_{\text{UNSAT}}$	UNSAT	iff	$\{\,\} \in F\|_J$ and $level(J) = 0$
(3)	$F :: J$	$\rightsquigarrow_{\text{DECIDE}}$	$F :: J, \dot{L}^l$	iff	$L \in atom(F\|_J) \cup \overline{atom(F\|_J)}$ and $l = level(J) + 1$
(4)	$F :: J$	$\rightsquigarrow_{\text{UNIT}}$	$F :: J, L^l$	iff	$\{L\} \in F\|_J$ and $l = level(J)$
(5)	$F :: J, \dot{L}^l, P$	$\rightsquigarrow_{\text{NB}}$	$F :: J, \overline{L}^{l-1}$	iff	$\{\,\} \in F\|_{J,\dot{L}^l,P}$

Figure 2: DPLL ARS

set of propositional variables AP. For better representation, we write $(F, J) \in R$ as $F :: J$. The evaluation of the set $F :: J$ is obtained by computing the reduct $F|_J$.

- The relation set $\rightarrow_{DPLL} = \{\rightsquigarrow_{\text{SAT}}, \rightsquigarrow_{\text{UNSAT}}, \rightsquigarrow_{\text{DECIDE}}, \rightsquigarrow_{\text{UNIT}}, \rightsquigarrow_{\text{NB}}\}$, and each relation is a reduction rule, given in Figure 2. We put a natural number in superscript which marks the level of each element in J. Here is a description of the reduction rules:
  1. *SAT rule:* is applicable if and only if all the clauses in F are satisfied.
  2. *UNSAT rule:* is applicable if and only if there is an empty clause in $F :: J$ and the level of the interpretation J is zero. We define level in the next rule description.
  3. *DECIDE rule:* chooses a free variable from $atom(F :: J)$ and its polarity, and appends it to the interpretation J. A variable $A \in atom(F)$ is a *free variable*, if and only if $A \in atom(F|_J)$, otherwise it is said to be *assigned*. The *level* of J, in symbols $level(J)$, is defined as the total number of decisions literals in J. The literal L appended to interpretation J by the DECIDE rule is called *decision literal*. We put a dot on the literal, i.e. $\dot{L}$, to represent a decision literal.
  4. *UNIT rule:* is the heart of the DPLL ARS, as the rule is used most of the time. This rule simply appends the literal appearing in a unit clause (unit clause in the reduct of the given CNF formula w.r.t. the interpretation), to the interpretation. This appended literal is called *propagated literal* under current interpretation J, and has the same $level(J)$. The unit rule is also called *unit propagation*.
  5. *NB rule:* stands for naive backtrack rule. This rule is applicable when there is a conflict in $F :: J$, i.e. there is a empty clause in $F :: J$, then the rule removes the last decision literal and the proceeding propagated literals from the interpretation. It is worth mentioning here is that P in the reduction does not contain any decision literal.

In practice, we use a preference order of the rules when there are more than one option of the applicability of the rules. Here is the preference order $\prec$, such that $a \prec b$ means a has less preference than b, of the applicability of the rules:

$$\rightsquigarrow_{\text{DECIDE}} \prec \rightsquigarrow_{\text{UNIT}} \prec \rightsquigarrow_{\text{NB}} \prec \rightsquigarrow_{\text{SAT}} \prec \rightsquigarrow_{\text{UNSAT}}$$

The reduction rules $\rightsquigarrow_{\text{SAT}}$ and $\rightsquigarrow_{\text{UNSAT}}$ are simple rules for satisfiable and unsatisfiable results respectively. They are also the termination rules which stop the application of any further reduction rules.

We define the notion of reason clause, that will be used in the sections to come. A clause C is *reason* for a literal L in $F :: J$ iff $C \in F$ and there exists an interpretation J_1, L, J_2 such that:

- $J_1, L, J_2 = J$ and

- $C|_{J_1} = \{L\}$

A reason clause C for a literal L means that L is a propagated literal due C. The set $reasons(F :: J)$ contains all the reason clauses for the literals in J, and is defined as:

$$reasons(F :: J) = \{C \in F \mid C \text{ is reason for literal } L \in s(J)\}$$

and is called the *set of reason clauses* of F w.r.t. the interpretation J.

Here is an example to explain how DPLL ARS works.

Example 2.12. Consider the following CNF formula:

$$F = \{\{1, 2\}, \{2, \neg 3\}, \{\neg 2, \neg 3, 4\}, \{\neg 1, 3\}, \{\neg 4\}\}$$

then the execution in DPLL ARS is:

	$F :: ()$	
$\rightsquigarrow_{\text{UNIT}}$	$F :: (4^0)$	$F\|_{(4^0)} = \{\{1, 2\}, \{2, \neg 3\}, \{\neg 2, \neg 3\}, \{\neg 1, 3\}\}$
$\rightsquigarrow_{\text{DECIDE}}$	$F :: (4^0, \dot{1^1})$	$F\|_{(4^0, \dot{1^1})} = \{\{2, \neg 3\}, \{\neg 2, \neg 3\}, \{3\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F :: (4^0, \dot{1^1}, 3^1)$	$F\|_{(4^0, \dot{1^1}, 3^1)} = \{\{2\}, \{\neg 2\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F :: (4^0, \dot{1^1}, 3^1, 2^1)$	$F\|_{(4^0, \dot{1^1}, 3^1, 2^1)} = \{\{\,\}\}$
$\rightsquigarrow_{\text{NB}}$	$F :: (4^0, \neg 1^0)$	$F\|_{(4^0, \neg 1^0)} = \{\{2\}, \{2, \neg 3\}, \{\neg 2, \neg 3\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F :: (4^0, \neg 1^0, 2^0)$	$F\|_{(4^0, \neg 1^0, 2^0)} = \{\{\neg 3\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F :: (4^0, \neg 1^0, 2^0, \neg 3^0)$	$F\|_{(4^0, \neg 1^0, 2^0, \neg 3^0)} = \{\,\}$
$\rightsquigarrow_{\text{SAT}}$	SAT	

The execution of DPLL ARS starts from $F :: (\,)$ (with empty interpretation). Following the preference, we set earlier, for the application of reduction rules, we apply the UNIT rule as there is a unit clause $\{4\}$ in F. No other rule applies except decide rule. We choose literal 1 as the decision literal. We apply unit rule twice because of the unit clause $\{3\}$ in $F|_{(4^0, \dot{1^1})}$ and $\{2\}$ in $F|_{(4^0, \dot{1^1}, 3^1)}$. This leads us to an empty clause in $F|_{(4^0, \dot{1^1}, 3^1, 2^1)}$. Now we apply naive backtrack to correct the last decision, and change the it from decision literal to propagated literal. A clause becomes a unit now in $F|_{(4^0, \neg 1^0)}$. Applying the unit rule three times, we get an empty set and now we can apply the SAT rule, which says that the given CNF formula F is satisfiable.

3. Modern SAT Solving

Most of the state-of-the-art SAT solvers are based on the simple DPLL algorithm, but they have undergone many refinements and new ideas over the last twenty years. We cover two prominent extensions of the DPLL algorithm: *conflict driven clause learning* (CDCL) in Section 3.1 and *lookahead* in Section 3.2. In Section 3.3, we briefly talk about preprocessing.

3.1. Conflict Driven Clause Learning

Conflict driven clause learning (CDCL) algorithm [SS96] is a major breakthrough in SAT solving, and as mentioned earlier, it is based on DPLL algorithm. CDCL gives the power to prune the search space early by learning new clauses, and as its name suggests, it learns the new clause whenever it encounters a conflict (conflict means that there is a empty clause in reduct of the given CNF formula w.r.t. given interpretation). Another feature that it brings with clause learning is the power to perform non-chronological backtracking.

We modify the DPLL ARS to explain the CDCL algorithm, resulting in the CDCL ARS. The two features of CDCL, clause learning and non-chronological backtracking when a conflict is encountered, is done by a single reduction rule. This rule is called the *CDBL rule.* The CDCL ARS is obtained by removing the NB rule from the $\rightarrow_{DPLL}$ and adding the reduction rule *CDBL rule*, i.e. $\rightsquigarrow_{\text{CDBL}}$. To explain the reduction CDBL rule, we first define the notion of *linear resolution derivation.* Given a CNF formula F and a clause C, such that $C \in F$, then a *linear resolution derivation* from C w.r.t. F is a sequence $S = (C_i \mid i \geq 0)$ of clauses defined inductively as follows:

- $C_0 = C$ and
- C_i is the resolvent of C_{i-1} and for some clause $E \in F$

If S is finite and C_n is the last element of the sequence, then it is called a *linear resolution derivation* from C to C_n w.r.t. F.

Definition 3.1. The **conflict driven backtrack learning** (CDBL) reduction rule is:

$$F :: J_1, \dot{L}^l, J_2 \rightsquigarrow_{\text{CDBL}} F \cup \{C_1\} :: J_1, L_1^{l-1}$$

iff there exists $C \in F$ such that $C|_{J_1,\dot{L},J_2} = \{\}$ and there is a linear resolution derivation from C to C_1 w.r.t. $reasons(F :: J_1, \dot{L}, J_2)$ and $C_1|_{J_1} = \{L_1\}$ and $l = level(J_1) + 1$. C_1 is called *learnt clause.*

The CDCL rule is applicable when reduct of F w.r.t. $J_1, \dot{L}, J_2$ has an empty clause (conflict), then the CDCL rule learns a new clause by performing a linear resolution derivation w.r.t. the reason clauses of $J_1, \dot{L}, J_2$, and add the new learnt clause to F. The backtracking information (interpretation J_1) is provided by the learnt clause C_1, such that the learnt clause becomes a unit clause in the reduct of $F \cup \{C_1\}$ w.r.t. J_1. The CDCL rule can learn different clauses depending on the number of reason clauses used in linear resolution derivation (this learning is often termed as learning schemes).

Most CDCL based SAT solver use a learning scheme called *first unique implication point* (in short *1-UIP*) [SS96]. We also use 1-UIP in our work, but we do not discuss learning schemes in this work. You can refer to [MSLM09] for details.

3.1.1. Decision Heuristic

DPLL algorithm does give us the DECIDE rule, but does not give much information about which literal we should choose. In this situation, heuristics help us out. Many heuristics have been proposed by researchers over the years, but one that is most successful and widely used is the *variable select independent decay sum* (VSIDS), originally proposed in [MMZ$^+$01] and later improved in [Rya04]. According to [KSMS11], VSIDS is the second most significant improvement after clause learning, in modern SAT solvers.

The idea of the VSIDS decision heuristic is to assign *activity* score to each variable based on its frequency in the formula, and this activity decays over time. Activity score of a variable is increased with usage of the variable in the linear resolution derivation of the learnt clause by the CDBL rule. Then the VSIDS picks the variable with the highest activity score.

VSIDS heuristic tells us which variable to choose, but for the DECIDE rule, we also need to know the polarity of the chosen variable. In modern SAT solvers, this information is given by *progress saving* heuristic [PD07]. The idea is to store the polarity information of variables when a backtracking is performed and choose the saved polarity for the chosen variable by the VSIDS heuristic. If no saved information, about polarity of a variable, is available then solver chooses negative polarity [Nik10]. This heuristic helps to avoid redoing work, which might get lost by non-chronological backtracking. By saving the last used polarity of a variable, solver usually finds solution faster [PD07].

3.1.2. Learnt Clauses Cleaning Policy

Modern SAT solvers use separate data structures for storing the given CNF formula (original problem) and the clauses learned by the CDBL rule. In practice we have limited memory in computers, so we can not keep all the learnt clauses in memory. To handle this problem, SAT solvers keep important learnt clauses and periodically (after certain number of conflicts) delete unimportant learnt clauses. They use different heuristics for measuring the importance of a learnt clause. Two widely used heuristics for learnt clause importance are: *activity* [GN02] and *LBD* [AS09].

Activity heuristic assigns an activity score to a learnt clause when it is learned. This score over time is increased geometrically if the clause is used in linear resolution derivation by the CDBL rule. According to this heuristic, a learnt clause has high importance, if its activity score is high, and a clauses with low activity score are deleted regardless of their size. SAT solver MiniSat [Nik10] uses the activity heuristic for learnt clauses.

Literal block distance (LBD) of a clause w.r.t. an interpretation, is the number of different decision levels of variables present in the clause. In practice, LBD heuris-

tic performs better than the activity heuristic. SAT solver GLUCOSE [AS09], which is based on MINISAT, uses LBD heuristic. GLUCOSE deletes learnt clauses more frequently than MINISAT does, because the authors of GLUCOSE claim that LBD is more accurate than activity heuristic, and deleting more clauses improves the performance of search and lowers the memory utilization. We now explain how the LBD of a learnt clause is calculated. Let C be a clause, J be an interpretation, and $s(J)$ be the set of elements present in J, then we define a function *lbd* that calculated the LBD of C w.r.t. J:

$$lbd(C) = |\{l, \text{ such that } L^l \in s(J) \text{ and } atom(L) \in atom(C)\}|$$

Here is an example of calculating LBD for a clause.

Example 3.2. Consider a clause $C = \{3, \neg 5, \neg 6\}$ and an interpretation $J = (1^0, \neg 3^0, \dot{4}^1, \dot{5}^2, \neg 6^2)$, then we have one literal in C of level 0, and two literals in C of level 2. In total, we have two different levels in the clause C, so LBD of C is two, $lbd(C) = 2$.

3.1.3. Restart Policy

Modern SAT solvers can efficiently solve the industrial SAT instances and we can see its evidence in the yearly SAT competitions. These industrial SAT instances exhibit a heavy-tailed phenomena [GSC97] [GSCK00], giving us the insight that different decision variable ordering lead to different solving time of a SAT solvers. Due to heavy-tailed phenomena, SAT solvers can have infinite median and infinite variance for solving time. Modern SAT solvers use restarts [GSK98], to bound the solving time median and variance. We add the *RESTART rule* in the CDCL ARS, and the rule is given by:

$$F :: J \rightsquigarrow_{\text{RESTART}} F :: (\ )$$

The RESTART rule removes every literal from the interpretation, but keeps the decision heuristic information and the learnt clauses. This way solver can benefit from the information (decision heuristic and learnt clauses) of the previous run (before performing restart) and may choose a different decision variable order. Restart policy is about deciding when to perform restart. We categorize restart policies into two: *static* and *dynamic*. We discuss here one example of each category.

The most widely used static restart policy is based on the *luby sequence* [LSZ93], that is:

$$(1, 1, 2, 1, 2, 4, 1, 2, 4, 8, 1, 2, 4, 8, 16, \ldots, \ldots)$$

MINISAT uses luby sequence with a constant factor (default is 100) for its restart policy, e.g. after 100 conflicts a restart is performed, second restart is after another 100 conflicts again, third restart is performed after 200 conflicts. The luby sequence guarantees that in a long run a large part of the search space is explored. This is due to the increasing behavior of the sequence.

GLUCOSE uses a dynamic restart policy [AS12b], that is based on LBD scores of a

certain number of last learnt clauses. GLUCOSE performs a restart if it is learning not so important clauses, i.e. clauses with high LBD scores. It maintains a global average of LBD scores, this average takes account of all the clauses learned so far since it started solving. We denote this global average by $avg_{\infty}LBD$. GLUCOSE keeps track of the LBD scores of the last X number of the learnt clauses, in a bounded queue and calculates running average of this bounded queue. We denote this average of bounded queue by $avg_{X}LBD$. *GLUCOSE* performs a restart, if

$$avg_{X}LBD * K > avg_{\infty}LBD$$

where K is a magic constant. Default values used by GLUCOSE are $K = 0.8$ and $X = 50$. The value of X is kept small for aggressive restart policy, so that the wrong decision made at top level can be corrected quickly. This dynamic restart policy shows good results on unsatisfiable instances, but not encouraging results on satisfiable instances, because of very aggressive restart policy it might the case the solver does not get much chance to grow the size of interpretation for satisfiable instance. For that reason, the authors of [AS12b] propose to postpone the restart if the solver has a good chance to assign values to all the variables, i.e. size of interpretation is equal to the number of variables in the given CNF formula F, in symbols, $|s(J)| = |atom(F)|$. They model this chance by recording the moving average of the size of interpretation at last five thousand conflicts, and if the size of the interpretation at that point of time is significantly greater than this average, then the next restart is postponed. We denote the moving average of size of interpretations by $avg_{5000}J$, then the condition to postpone a restart is given by:

$$|s(J)| > R * avg_{5000}J$$

where R is the significance parameter and the default value used by GLUCOSE for this parameter is $R = 1.4$.

SAT solvers do not throw away the learnt clauses while performing a restart, as similar conflicts may arise again which lead to the learning of these learnt clauses. They also use the same decision heuristic values before performing a restart, because decision heuristics evolve with time and the solver can benefit from that.

3.2. Lookahead

Lookahead SAT solvers are also based on the DPLL algorithm, but the major difference from a CDCL SAT solvers is that lookahead SAT solver are heavily driven by expensive decision heuristics. We discuss here some of these heuristics, but for more detailed view on lookahead, we suggest [HvM09] for reading. We look at some definitions that are important for further understanding.

The *lookahead* on $F :: J$ with respect to L, in symbols $lookahead(F :: J, \dot{L})$, is defined as:

$$lookahead(F :: J, \dot{L}^{l}) = \begin{cases} \top & \text{, if } \{\ \} \in F|_{J,\dot{L}^{l},P} \text{ and } F :: J, L \rightsquigarrow^{*}_{\text{UNIT}} F :: J, \dot{L}^{l}, P \\ \bot & \text{, if otherwise} \end{cases}$$

where P contains propagated literals only. Literal L is called a *failed literal* in $F :: J$ if $lookahead(F :: J, \dot{L}) = \top$, i.e. a conflict is found. The main idea of lookahead SAT solvers is to perform *lookahead* on some interesting free variables, find failed literals and choose a decision literal according to some heuristics (explained later in this section). We call this process *lookaheadDecide*. To utilize the time efficiently, it performs some reasoning techniques (also explained later in this section). For explaining the Lookahead ARS, we modify the set R_{DPLL} used in DPLL ARS (on page 8). Now each clause C of F has a natural number in the superscript, which denotes its level, i.e. the decision level at which C is locally learned (explained later in Section 3.2.3).

Definition 3.3. Lookahead ARS $(R_{LA}, \rightarrow_{LA})$ is defined as:

$$R_{LA} = R_{DPLL}$$

$$\rightarrow_{LA} = \{\rightsquigarrow_{\text{SAT}}, \rightsquigarrow_{\text{UNSAT}}, \rightsquigarrow_{\text{DECIDE}}, \rightsquigarrow_{\text{UNIT}}, \rightsquigarrow_{\text{LA\_BACK}}, \rightsquigarrow_{\text{LR}}, \rightsquigarrow_{\text{NB}}\}$$

The reduction rules of **Lookahead ARS** are shown in Figure 3. The first four rules in the figure are same as DPLL ARS, see explanation of these rules on page 8.

5. *LA_BACK rule:* stands for lookahead back, and is used by the procedure *lookaheadDecide* only, heuristic to choose decision literal. This rule gives the power to go back even without a conflict, that is required by *lookaheadDecide*.

6. *LR rule:* stands for local reasoning rule. This is also used by the procedure *lookaheadDecide* only, and it adds clauses to the CNF formula based on some reasoning techniques, we discuss these techniques later in this section. These added clauses are called local learned clauses, which are locally valid at level l (means they are valid as long as $level(J) \geq l$).

7. *NB rule:* is a extended version of the NB rule from DPLL ARS; apart from correcting the last decision, this extended version also deletes the locally learnt clauses whose level is greater than the level of interpretation J.

Remark. For simplicity, we group together the clauses of same level into a CNF-Formula, so F^0 denotes clauses of original problem and clauses learnt at level zero while F^2 denotes the clauses learnt at level two. You should note that this grouping is only for Lookahead ARS.

(1)	$F :: J \rightsquigarrow_{\text{SAT}} F ::$ SAT iff $F\vert_J = \{\,\}$
(2)	$F :: J \rightsquigarrow_{\text{UNSAT}} F ::$ UNSAT iff $\{\,\} \in F\vert_J$ and $level(J) = 0$
(3)	$F :: J \rightsquigarrow_{\text{DECIDE}} F :: J, \dot{L}^l$ iff $L \in atom(F\vert_J) \cup \overline{atom(F\vert_J)}$ and $l = level(J) + 1$
(4)	$F :: J \rightsquigarrow_{\text{UNIT}} F :: J, L^l$ iff $\{L\} \in F\vert_J$ and $l = level(J)$
(5)	$F :: J, \dot{L}, P \rightsquigarrow_{\text{LA\_BACK}} F :: J$ iff $\{\,\} \notin F\vert_{J,\dot{L},P}$
(6)	$F :: J \rightsquigarrow_{\text{LR}} F, F^l :: J$ iff $F\vert_J \models F^l\vert_J$ and $level(J) = l$
(7)	$F^0, \ldots, F^l :: J, \dot{L}, P \rightsquigarrow_{\text{NB}} F^0, \ldots, F^{l-1} :: J, \overline{L}$ iff $\{\,\} \in (F^0 \cup \ldots \cup F^l)\vert_{J,\dot{L},P}$

Figure 3: Lookahead ARS

3.2.1. Lookahead Decision Heuristics

We discuss now the procedure *lookaheadDecide*, that is used by lookahead SAT solvers for choosing a decision literal. The most simple and popular decision heuristic which is still used in lookahead SAT solvers, is given in [Fre95]. This decision heuristic chooses a variable which gives the most simplest sub-problems. One way for choosing such a variable is to first calculate the *difference* (in short $diff$), each polarity of that variable makes to a given problem, by performing one-step lookahead, and then combining these two $diff$ scores. This combined $diff$ score is called $mixdiff$ [HvM09]. There are different variations of $diff$ based on how it is calculated. One such calculation is given in [Fre95], that counts the number of assigned variables, we call this heuristic as $diff1$. Given a CNF formula F and a literal L, $diff1$ calculates the number of assigned variables by performing *lookahead* on F with respect to L, if L is not a failed literal. Mathematically $diff1$ is defined as:

$$diff1(F :: J, \dot{L}) = \begin{cases} |atom(F) - atom(F|_{J,\dot{L},P})| & \text{, if } F :: J, \dot{L} \rightsquigarrow^*_{\text{UNIT}} F :: J, \dot{L}, P \\ & \quad \text{and } lookahead(F :: J, \dot{L}) = \bot \\ 0 & \text{, if } lookahead(F :: J, \dot{L}) = \top \end{cases}$$

Another variation is to calculate the number of new binary clauses, denoted by $diff2$.

$$diff2(F :: J, \dot{L}) = \begin{cases} |\{C \text{ such that } |C| = 2 \text{ and} \\ \quad C \notin F|_J \text{ and} \\ \quad C \in F|_{J,\dot{L},P}\}| & \text{, if } lookahead(F, \dot{L}) = \bot \text{ and} \\ & \quad F :: J, \dot{L} \rightsquigarrow^*_{\text{UNIT}} F :: J, \dot{L}, P \\ 0 & \text{, if } lookahead(F, \dot{L}) = \top \end{cases}$$

I have used a mixture of both $diff1$ and $diff2$ in my previous work [Irf12], we denote this mixed heuristic by $diff3$, which is defined as:

$$diff3(F :: J, \dot{L}) = 0.3 * diff1(F :: J, \dot{L}) + 0.7 * diff2(F :: J, \dot{L})$$

Given a CNF-Formula F, an interpretation J, and a literal $L = A \in atom(F|_J)$, we define $mixdiff$ as:

$$\begin{aligned} mixdiff(F :: J, A) =& 1024 * diff(F :: J, \dot{L}) * diff(F :: J, \dot{\overline{L}}) + diff(F :: J, \dot{L}) \\ & + diff(F :: J, \dot{\overline{L}}) \end{aligned}$$

With the definition of $mixdiff$, we define decision heuristic as choosing the variable which gives the maximum score $mixdiff$ score, mathematically we represent as:

$$lookaheadDecide(F :: J) = arg\ max_{A \in atom(F|_J)} mixdiff(F :: J, A)$$

The decision heuristic *lookaheadDecide* gives us a variable, but the DECIDE rule in the Lookahead ARS requires also the polarity of the chosen variable. A simple way

is to choose randomly, but that is often not how its done in lookahead SAT solvers. We now look what different lookahead SAT solvers do for choosing the polarity.

Lookahead SAT solver SATZ [LA97] always chooses positive polarity, another solver KCNF [DD04] chooses the polarity which has higher frequency of the chosen decision variable in the formula. Lookahead SAT solver MARCH [HvM06] chooses the polarity for which the $diff$ score is lower. The reason given by author for choosing polarity with lower $diff$ score is that the lower $diff$ score assigns less number of variables than the the higher $diff$ score does, thus has lower probability of making mistake and higher probability of leading to a satisfiable solution; but on the other hand, polarity with higher $diff$ score can lead to less computation than polarity with lower $diff$ score.

3.2.2. Pre-selection Heuristic

Pre-selecting small number of variables to be used by *lookaheadDecide* can reduce its computational cost. On the other hand, there is also a chance to degrade the overall performance of a solver if the pre-selected variables do not contain the optimal variable (the variable which would have been chosen by the decision heuristic without performing the pre-selection of variables). Due to these reasons, pre-selection of the variables is a crucial step. We discuss here one heuristic that we use for pre-selection of variables, called *recursive weighted heuristic*.

Recursive weighted heuristic is presented in [MdWH10] as a decision heuristic for CNF formulas of maximum clause size three, and [AF10] extends this heuristic for CNF formulas of arbitrary size. We use this heuristic for pre-selecting variables, because it can be cheaply computed. Recursive weighted heuristic is an iterative model and accuracy of the heuristic increases with the number of iterations performed. We present here the basic idea of this heuristic. The heuristic value $h_i(L)$ means the tendency of the literal L being an element of the model for a given $F :: J$.
For each $A \in atom(F|_J)$:

$$h_0(A) = h_0(\neg A) = 1$$

For each $L \in atom(F|_J) \cup \overline{atom(F|_J)}$:

$$h_{i+1}(L) = \sum_{C \in F|_J} \left(\frac{\gamma^{k-|C|}}{\mu_i^{|C|-1}} \prod_{L_1 \in C \setminus \{L\}} h_i(\overline{L_1})\right)$$

$$\mu_i = \frac{1}{2 * |atom(F)|} \sum_{A \in atom(F)} (h_i(A) + h_i(\neg A))$$

where k is the maximum clause size, γ is the importance constant which is set to 5 by [AF10] and μ_i is the average heuristic value in iteration i. The importance constant γ is used to give more weight to shorter clauses and the average heuristic value μ_i to normalize the scores. The heuristic value for a literal is treated as $diff$ score and $mixdiff$ is used to calculate the value of a free variable in F, and these free variables are sorted in descending according to their $mixdiff$ and then we can pick some of the top variables. Usually, SAT solvers like *march* pick top 10% of the

free variables, this strategy is called $rank_{10\%}$, but choosing always a static number of pre-selection variables does not seem to be efficient, and [HDvZvM05] provides a correlation between the number of pre-selection variables and number of failed literals. They suggest to choose bigger number of pre-selection variables, when the solver finds more failed literals. They provide a adaptive way of selecting this number, and call this heuristic *adaptive ranking*, which is given as:

$$RankAdapt_n = \alpha + \frac{\beta}{n} \sum_{i=1}^{n} \#failed_i$$

where α is the lower bound for number of pre-selected variables, β is a constant factor which models the importance of failed literals, and $\#failed_i$ denotes the number of failed literals found while *lookaheadDecide* chooses i^{th} decision literal, and n is the total number of decision made by *lookaheadDecide*. The authors choose $\alpha = 5$ and $\beta = 7$, based on experiments.

3.2.3. Local Learning

The decision heuristic *lookaheadDecide* is computationally expensive process and to get most out of it, lookahead SAT solvers apply some reasoning techniques during that process. We explain here some of these techniques.

Let F be a CNF formula, J be an interpretation, and L be a literal in one of the clause in $F|_J$, then when we perform $lookahead(F :: J, \dot{\overline{L}})$, some literals e.g. L_1 may be propagated by the UNIT rule. This means that the literal L_1 is implied by the literal L and these relationships between two literals called either *direct implications* or *indirect implication* otherwise. By direct implication, we mean that there exists a clause $C = \{\overline{L}, L_1\}$ in the CNF formula F, and indirect implication if C does not exist in F. We save the indirect implications as binary clause, and assign a level of the interpretation J to this binary clause. This addition of binary clauses is called *local learning* [HvM09]. Consider the $level(J) = l$, then the indirect implications can be added as binary clause e.g. $\{\overline{L}, L_1\}^l$ to the CNF formula F. As the name suggests, these clauses are not globally valid and they must be removed when backtracking, i.e. when level becomes less than l. The next example will make clear this property. Before, we require the definition of iterative unit propagation. Given $F :: J$, the function *iterative unit propagation* after deciding L, $iup(F :: J, \dot{L})$ is defined as:

$$iup(F :: J, \dot{L}) = s(P) \qquad \text{if } F :: J, \dot{L} \rightsquigarrow^*_{\text{UNIT}} F :: J, \dot{L}, P$$

where $s(P)$ denotes the set of propagated literals present in P. Here is an example to explain the difference between direct implications and indirect implications.

Example 3.4. Consider the following CNF formula:

$$F = \{\{\neg 1, 2\}, \{\neg 1, \neg 2, 3\}, \{\neg 1, \neg 3, 4\}, \{1, 3, 6\}, \{\neg 1, 4, \neg 5\}, \{1, \neg 6\}, \{4, 5, 6\}, \{5, \neg 6\}\}$$

as interpretation J is empty, so $level(J) = 0$ and $F^0 := F$. We apply *lookahead* on one of the literal, say 1, so *lookahead* chooses 1 as decision literal and applies unit propagation:

$$
\begin{array}{lll}
 & F^0 :: (\,) & \\
\rightsquigarrow_{\text{DECIDE}} & F^0 :: (\dot{1}^1) & F^0|_{(\dot{1}^1)} = \{\{2\},\{\neg 2,3\},\{\neg 3,4\},\{4,\neg 5\},\\
 & & \qquad \{4,5,6\},\{5,\neg 6\}\} \\
\rightsquigarrow_{\text{UNIT}} & F^0 :: (\dot{1}^1, 2^1) & F^0|_{(\dot{1}^1,2^1)} = \{\{3\},\{\neg 3,4\},\{4,\neg 5\},\{4,5,6\},\\
 & & \qquad \{5,\neg 6\}\} \\
\rightsquigarrow_{\text{UNIT}} & F^0 :: (\dot{1}^1, 2^1, 3^1) & F^0|_{(\dot{1}^1,2^1,3^1)} = \{\{4\},\{4,\neg 5\},\{4,5,6\},\{5,\neg 6\}\} \\
\rightsquigarrow_{\text{UNIT}} & F^0 :: (\dot{1}^1, 2^1, 3^1, 4^1) & F^0|_{(\dot{1}^1,2^1,3^1,4^1)} = \{\{5,\neg 6\}\} \\
\rightsquigarrow_{\text{LA\_BACK}} & F^0 :: (\,) &
\end{array}
$$

so, $iup(F^0 :: (\dot{1}^1)) = \{2,3,4\}$ and the implication clauses are $\{\neg 1,2\}$, $\{\neg 1,3\}$, $\{\neg 1,4\}$. The first implication clause is a direct implication while second and third are indirect implications.

One problem with the local learned clauses is that they are too many (more than the original number of clauses in the CNF-Formula) and they can degrade the performance of unit propagation. A computationally cheap solution to this problem is detection of *necessary assignments*, inspired from Stålmarcks's proof procedure [SS98] for propositional logic.

Definition 3.5. Given $F :: J$ and we apply *lookahead* on literal L, then L_1 is a **necessary assignment** if and only if:

$$L_1 \in iup(F :: J, \dot{L}) \cap iup(F :: J, \dot{\overline{L}})$$

The detected necessary assignment can be added as a local learnt clause, we provide to following example to elaborate that:

Example 3.6. Consider the CNF formula from the Example 3.4, we start from where we finished it.

$$F = \{\{\neg 1,2\},\{\neg 1,\neg 2,3\},\{\neg 1,\neg 3,4\},\{1,3,6\},\{\neg 1,4,\neg 5\},\{1,\neg 6\},\\ \{4,5,6\},\{5,\neg 6\}\}$$

and $iup(F^0 :: (\dot{1})) = \{2,3,4\}$.
Now applying lookahead on $\neg 1$, i.e. choosing $\neg 1$ as the decision literal and applying unit propagation:

$$
\begin{array}{lll}
 & F^0 :: (\,) & \\
\rightsquigarrow_{\text{DECIDE}} & F^0 :: (\neg\dot{1}^1) & F^0|_{(\neg\dot{1}^1)} = \{\{3,6\},\{\neg 6\},\{4,5,6\},\{5,\neg 6\}\} \\
\rightsquigarrow_{\text{UNIT}} & F^0 :: (\neg\dot{1}^1, \neg 6^1) & F^0|_{(\neg\dot{1}^1,\neg 6^1)} = \{\{3\},\{4,5\}\} \\
\rightsquigarrow_{\text{UNIT}} & F^0 :: (\neg\dot{1}^1, \neg 6^1, 3^1) & F^0|_{(\neg\dot{1}^1,\neg 6^1,3^1)} = \{\{4,5\}\} \\
\rightsquigarrow_{\text{LA\_BACK}} & F^0 :: (\,) &
\end{array}
$$

so, $iup(F^0 :: (\neg\dot{1})) = \{3,\neg 6\}$ and the literal 3 is a necessary assignment, because

it 3 is present in both $iup(F^0 :: (\neg\dot{1}))$ and $iup(F^0 :: (\dot{1}))$.
We apply the LR rule to add necessary assignment as a local learnt clause.
$\rightsquigarrow_{\text{LR}} \quad F^0, \{3\}^0 :: (\ )$
We have assigned level zero to the local learnt clause because the level of interpretation is zero.

Another reasoning technique is to find equivalent literals, and add two binary implication clauses to local learnt clauses – one for each direction of equivalence. This technique is called *equivalence reasoning* [Li03].

Definition 3.7. Given $F :: J$, we perform lookahead on L_1 and $\overline{L_1}$, then L_1 and L_2 are **equivalent literals** if:

$$L_2 \in iup(F :: J, \dot{L_1}) \cap \overline{iup(F :: J, \dot{\overline{L_1}})}$$

Here is an example of equivalent literals.

Example 3.8. Consider the following CNF formula:

$$F = \{\{\neg 1, 2\}, \{\neg 1, 4\}, \{1, 5\}, \{\neg 2, 3, \neg 4\}, \{\neg 3, \neg 5, 6\}, \{\neg 5, \neg 6\}\}$$

Applying lookahead on literal 1:

	$F^0 :: (\ )$	
$\rightsquigarrow_{\text{DECIDE}}$	$F^0 :: (\dot{1^1})$	$F\vert(\dot{1^1}) = \{\{2\}, \{4\}, \{\neg 2, 3, \neg 4\}, \{\neg 3, \neg 5, 6\}, \{\neg 5, \neg 6\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1^1}, 2^1)$	$F\vert(\dot{1^1}, 2^1) = \{\{4\}, \{3, \neg 4\}, \{\neg 3, \neg 5, 6\}, \{\neg 5, \neg 6\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1^1}, 2^1, 4^1)$	$F\vert(\dot{1^1}, 2^1, 4^1) = \{\{3\}, \{\neg 3, \neg 5, 6\}, \{\neg 5, \neg 6\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{1^1}, 2^1, 4^1, 3^1)$	$F\vert(\dot{1^1}, 2^1, 4^1, 3^1) = \{\{\neg 5, 6\}, \{\neg 5, \neg 6\}\}$
$\rightsquigarrow_{\text{LA\_BACK}}$	$F^0 :: (\ )$	

So, $iup(F :: (\dot{1})) = \{2, 3, 4\}$, now applying lookahead on literal $\neg 1$:

	$F^0 :: (\ )$	
$\rightsquigarrow_{\text{DECIDE}}$	$F^0 :: (\dot{\neg 1^1})$	$F\vert(\dot{\neg 1^1}) = \{\{5\}, \{\neg 2, 3, \neg 4\}, \{\neg 3, \neg 5, 6\}, \{\neg 5, \neg 6\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{\neg 1^1}, 5^1)$	$F\vert(\dot{\neg 1^1}, 5^1) = \{\{\neg 2, 3, \neg 4\}, \{\neg 3, 6\}, \{\neg 6\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{\neg 1^1}, 5^1, \neg 6^1)$	$F\vert(\dot{\neg 1^1}, 5^1, \neg 6^1) = \{\{\neg 2, 3, \neg 4\}, \{\neg 3\}\}$
$\rightsquigarrow_{\text{UNIT}}$	$F^0 :: (\dot{\neg 1^1}, 5^1, \neg 6^1, \neg 3^1)$	$F\vert(\dot{\neg 1^1}, 5^1, \neg 6^1, \neg 3^1) = \{\{\neg 2, \neg 4\}\}$
$\rightsquigarrow_{\text{LA\_BACK}}$	$F^0 :: (\ )$	

$iup(F :: (\neg\dot{1})) = \{5, \neg 6, \neg 3\}$
We get 1 and 3 as equivalent literals, because $3 \in iup(F :: (\dot{1})) \cap \overline{iup(F :: (\neg\dot{1}))}$.
This equivalence between literals 1 and 3 can be saved as local learnt clauses.
$\rightsquigarrow_{\text{LR}} \quad F^0, \{\neg 1, 3\}^0, \{1, \neg 3\}^0 :: (\ )$
We have assigned level zero to the local learnt clauses because the level of interpretation is zero.

3.2.4. Double Lookahead

The idea of *double lookahead* is to check if a literal gets a high $diff$ score, because it might be the case that this literal leads the search to a conflict. This check is done by performing another lookahead, as also indicated by the name. The idea of double lookahead is given in [Li99].

Definition 3.9. Given $F :: J, \dot{L}_1, P$ then ***doubleLookahead*** is:
$doubleLookahead(F :: J, \dot{L}_1, P)$

$$= \begin{cases} \top & \text{,if } lookahead(F :: J, \dot{L}_1, P, \dot{L}_2) = \top \text{ and} \\ & \quad lookahead(F :: J, \dot{L}_1, P, \overline{\dot{L}_2}) = \top \\ & \quad \text{for some } L_2 \in atom(F|_{J,\dot{L}_1,P}) \cup \overline{atom(F|_{J,\dot{L}_1,P})} \\ \bot & \text{,otherwise} \end{cases}$$

Double lookahead is successful (denoted by $\top$ in definition) if it finds conflict on both polarities of a variable, otherwise unsuccessful (denoted by $\bot$). Important point is to know when to perform double lookahead, because *lookaheadDecide* is already expensive w.r.t. computation, so we need some heuristic to tell the solver when to perform double lookahead. We discuss one heuristic [HvM07] that is used in this work, double lookahead is performed on $F :: J, \dot{L}$ if:

$$diff(F :: J, \dot{L}) > trigger$$

Value of $trigger$ initialized with zero, and is updated to $diff$ score value of L, if double lookahead is not successful, otherwise the value is not changed. The value of $trigger_2$ is slightly reduced each time when *lookaheadDecide* chooses a decision literal. Similar to lookahead, solver performs double lookahead on some interesting free variables (pre-selected variables). While performing double lookahead, solver can detect some failed literals and necessary assignments, which can be saved as binary learnt clauses and these clauses are called *double lookahead resolvents*. For example, consider that the double lookahead is performed on $F :: J, \dot{L}_1, P$ and L_2 is detected as necessary assignment, then a binary clause $\{\overline{L_1}, L_2\}^l$ can be added to the CNF formula F, where l is the level of the learnt clause if $level(J) = l$.

3.3. Preprocessing

Most of the modern SAT solver, that take part in the *SAT Competitions*, use preprocessor before they start solving. The job of preprocessor is to simply the input CNF formula by applying some very fast techniques; time used to do preprocessing is usually negligible compared to overall solving time. The techniques used by preprocessors can be divided in to groups: *model preserving* and *non-model preserving*. As the name suggest, model preserving techniques preserve the model of the formula, while non-model preserving techniques do not.

We will use SatELite preprocessor [EB05] in our work; the techniques used by this preprocessor are used in almost all modern preprocessor for SAT solvers. SatELite uses techniques to remove variables and clauses from the input CNF formula. It

has been shown in [EB05] that simplifying CNF formula by removing variables and clauses decreases the solving time of a SAT solver. Solvers get improvement by using preprocessor before starting the search is due to the speedup in unit propagation (less clauses to check for unit propagation) and less variable for making decisions.

As preprocessing of a CNF formula is not focus of this work, we do not discuss its details. Nevertheless, we use preprocessor for comparing our solver with other state-of-the-art solvers.

4. Parallel SAT Solving

In this section, we introduce different ways of parallelisms for SAT solving. We make the introduction to the parallelism short and focus more on our approach for parallel SAT solving. For a detailed overview of parallel SAT solving approached, we suggest two readings, [Sin06] and [HMN+11]. Before we start discussing the different ways of parallelism for SAT solving, we first introduce some notations.

Task is referred to some job that needs to be executed, i.e. solving a CNF formula. We call an instance of a solver that solves a CNF formula, an *incarnation* of a solver. Now, we define speedup and efficiency for parallelism.

Definition 4.1. Let T_s and T_p be the solving time of sequential and parallel algorithm, respectively; let k be the processing units for parallel algorithm. Then **speedup** S is the ratio of the sequential solving time to parallel solving time:

$$S = \frac{T_s}{T_p}$$

and **efficiency** E is defined as speedup divided by the number of processing units (number of available resources):

$$E = \frac{S}{n} = \frac{T_s}{T_p * k}$$

we call the speedup of parallel algorithm as *superlinear speedup*, if $S > k$ or $E > 1$.

We give a very basic knowledge of computer architecture here. A *central processing unit* (CPU) is the hardware in a computer that carries out the computer instructions. The number of tasks that can be run in parallel in a CPU is equal to the number of *cores* inside CPU. There is a connection between CPU and *main memory*. There are multi-CPU systems, which have multiple CPUs connected on the same board, and each CPU has a private connection to main memory. One CPU can also access the data from another CPU memory (remote), in a *non-uniform* way, and the access time of remote memory is greater than the access time of private memory. There are also multi-core systems, which have multiple cores in a CPU, and each core shares the connection to main memory. Unlike multi-CPU systems, the memory access in multi-core systems is uniform.

Parallel SAT solvers have been build for the last twenty years, at a slower pace than the sequential SAT solvers, but due to the technological shift from single-core to multi-core CPUs, attention to parallel SAT solving has increased for the last couple of years. We see more than one core CPUs in different machines, like servers, desktops, laptops, smart phone, etc. Our interest is build parallel SAT solver for a multi-core system environment, so we will focus on multi-core systems. In this section and the sections to follow, we assume the environment to be a multi-core systems environment, whenever we talk about parallel SAT solvers.

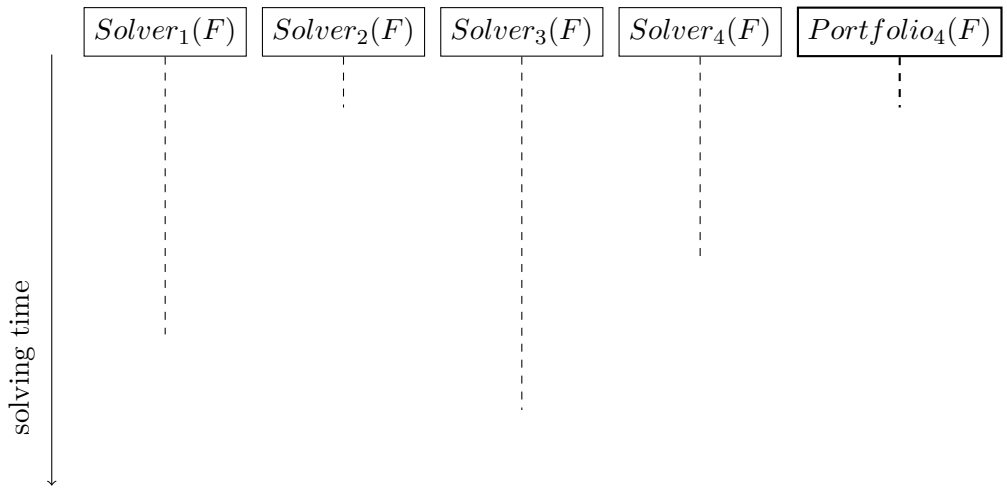


Figure 4: Example - Competitive Parallelism

4.1. Competitive Parallelism vs Cooperative Parallelism

In **competitive parallelism,** we simply run multiple incarnations on the same CNF formula in a competitive environment, i.e. we get the answer when the first incarnation solves the CNF formula. We call the parallel SAT solvers, which use competitive parallelism, as *portfolio solvers.* Figure 4 shows an example of a portfolio solver: $Solver_1$, $Solver_2$, $Solver_3$, $Solver_4$ are different incarnations which have different solving time of the same CNF formula F. We represent the solving time of each incarnation with a dashed line and it is clear from the figure that $Solver_2$ finishes earlier than the other solvers. By making portfolio, shown in the figure as $Portfolio_4$ of the solvers $Solver_1$, $Solver_2$, $Solver_3$, $Solver_4$, we solve the CNF formula F (satisfiable or unsatisfiable) with the best solving time among these solvers. We can formulate the solving time property of portfolio solvers in the following proposition.

Proposition 4.2. *Let k be the number of incarnations that can run in parallel and T_i be the solving time of incarnation $Solver_i$, then the expected solving time T_{pfs} of a portfolio solver with k incarnations, is:*

$$T_{pfs} = arg_{1 \leq i \leq k}\ min\{T_i\}$$

Since the success of ManySAT [HJS08], the research in parallel SAT solving has been focused on competitive parallelism mostly: different approaches to share clauses have been explored in [HS09] [HJS09a] [AHJ+12], concepts of diversification and intensification [HS04] in the search has been investigated in [GHJS10]. We will discuss the necessary details of these improvements in this section and Section 5.

On the other hand, in **cooperative parallelism**, we split the search space of the CNF formula such that incarnations solve different parts of the search space. Splitting of search space is done by a *partitioning function.* We call the parallel solvers, which use cooperative parallelism, as *search space partitioning solvers.* We provide a very

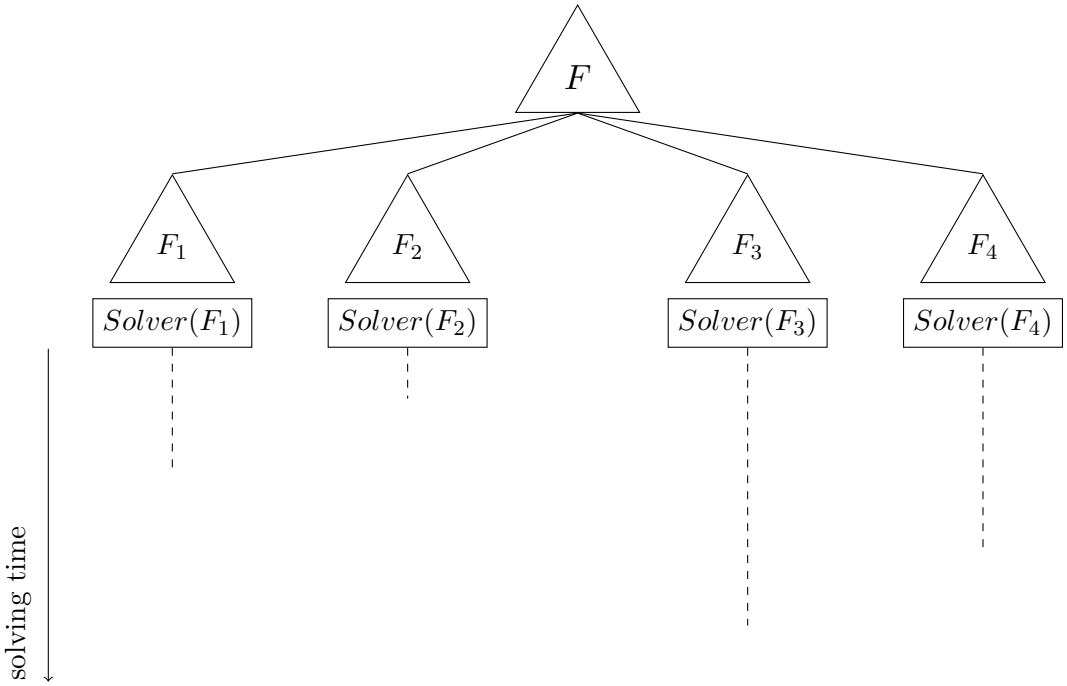


Figure 5: Example - Cooperative Parallelism

basic definition of partitioning function.

Definition 4.3. Given a CNF formula F, a partitioning function (pf) is defined as a function that takes F as input and returns a set of CNF formulas. In symbols, we represent pf as:

$$pf(F) = \{F_1, F_2, \ldots, F_n\}$$

such that:

- F is satisfiable if there exists $F_i \in pf(F)$ is satisfiable
- F is unsatisfiable if every $F_i \in pf(F)$ is unsatisfiable

Figure 5 shows an example of a search space partitioning solver: having four partitions F_1, F_2, F_3, F_4, and a incarnation *Solver* solves a partition (we will explain in next subsection that how to produce partitions). Unlike competitive parallelism, cooperative parallelism has different cases depending upon the CNF formula begin satisfiable or unsatisfiable. We consider the search space partitioning solver shown in Figure 5, and suppose that the CNF formula F is satisfiable, then the solving time of the search space partitioning solver is equal to the solving time of the first incarnation that solves the partition with result satisfiable, e.g. suppose $Solver(F_2)$ finishes first with result unsatisfiable and $Solver(F_1)$ finishes second but with result satisfiable, then the search space partitioning solver stops the incarnation $Solver(F_3)$ and $Solver(F_4)$, and the solving time of the solver is equal to the solving time of

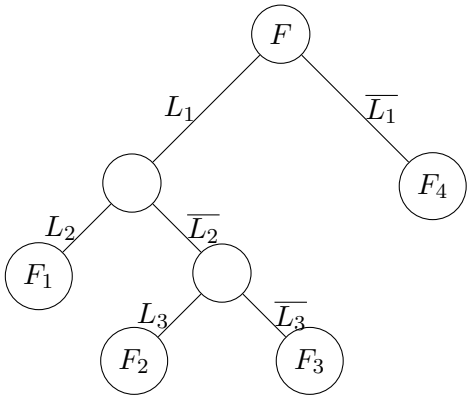


Figure 6: Example - Simple Method for Creating Partitions

$Solver(F_1)$. In case of a CNF formula that is unsatisfiable, all the incarnations should return result unsatisfiable, and the solving time of the search space partitioning solver is equal to the incarnation which finishes last, e.g. in Figure 5, incarnation $Solver(F_3)$ finishes last, so the solving time of the solver is equal to solving time of $Solver(F_3)$. We put this property of cooperative parallelism into the following proposition.

Proposition 4.4. *Let k be the number of incarnation that can run in parallel, and $F_1, F_2, \ldots, F_k$ be the search partitions of the CNF formula F such that an incarnation Solver solves each a partition. For $1 \leq i \leq k$, let $Solver(F_i)$ represent the incarnation solving partition F_i and T_i be the solving time of $Solver(F_i)$, then the solving time of search space partitioning solver, represented by T_{ssp}, is:*

$$T_{sps} = \begin{cases} arg_{1 \leq i \leq k}\ min\{T_i \mid Solver(F_i) \rightsquigarrow SAT\} \\ \qquad\qquad \text{, if } F \text{ is satisfiable instance and } \exists Solver(F_i) \rightsquigarrow SAT \\ arg_{1 \leq i \leq k}\ max\{T_i \mid Solver(F_i) \rightsquigarrow UNSAT\} \\ \qquad\qquad \text{, if } F \text{ is unsatisfiable instance and } \forall Solver(F_i) \rightsquigarrow UNSAT \end{cases}$$

4.2. Parallel SAT Solving with Search Space Partitioning

We now show how we can make a search space partitioning SAT solver. The first point to consider is to create partitions, that we discuss in Section 4.2.1. The idea is to constraint the CNF formula by adding clauses to it, we call these additional clauses as *partitioning constraints.* Then comes the part of solving these partitions, we discuss in Section 4.2.2 and there we also introduce *the plain partition* and *the iterative partitioning.* We discuss the iterative partitioning in more detail, because our focus is to use iterative partitioning in this work. Although sharing information among the partitions is optional, but it has been shown by researchers that sharing information, like learnt clauses, can improve the performance of the solver. We discuss about sharing information in Section 4.2.3.

4.2.1. Creating Partitions

We create simple partitions from a search tree of arbitrary height, not necessarily a balanced tree, such that the remaining search space of a leaf makes a partition. We create this search tree by splitting on a complementary pair of literals. We call this method **simple method** for creating partitions. We explain the simple method with the help of the following example.

Example 4.5. We consider a CNF formula F for which we want to create search partitions. Figure 6 shows the search tree of F. We split the root node, F, on complementary pair of literals L_1 and $\overline{L_1}$, the literals are shown on top of the edges. The node that we created by choosing literal L_1, we split it on complementary pair of literals L_2 and $\overline{L_2}$, and similarly we split the node, created by choosing literal $\overline{L_2}$, on complementary pair of literals L_3 and $\overline{L_3}$. Now we create a partition for each path from root to leaf, by adding the literals on the paths as unit clauses in the CNF formula F. In the figure, we use F_1, F_2, F_3, F_4 for partitions, that are defined as:

$$
\begin{aligned}
F_1 &= F \cup \{\{L_1\}, \{L_2\}\} \\
F_2 &= F \cup \{\{L_1\}, \{\overline{L_2}\}, \{L_3\}\} \\
F_3 &= F \cup \{\{L_1\}, \{\overline{L_2}\}, \{\overline{L_3}\}\} \\
F_4 &= F \cup \{\{\overline{L_1}\}\}
\end{aligned}
$$

For $1 \leq i \leq 4$, $F_i \setminus F$ are the *partitioning constraints*.

You should note that, in the earlier example, we use a unbalanced search tree and the decision to choose a node that splits is not explained; there are heuristics for choosing node. For further reading about heuristics for simple method, you can refer to [Irf12]. Nevertheless, you can also use a balanced search tree for creating partitioning using simple method. Another point to note here is that a leaf node, in the search tree, can be a conflicting node, i.e. the reduct of CNF formula F w.r.t. the literals on the path from root to the leaf node (these literals can be treated as interpretation) contains an empty clause. Due to this scenario, we can not predict the number of partitions that are created with the simple method.

A different approach called **scattering method**, for creating partitions, is presented in [HJN06]. The scattering method uses a biased search tree, that splits a node on a complementary pair of *cubes* D_1 and $\overline{D_1}$. A **cube** [HKWB12] is a set of unit clauses. We define the complement of cube as follows. Given a cube $D = \{\{L_1\}, \{L_2\}, \ldots, \{L_c\}\}$, then we define a the function $cube2clause(D)$ that returns a clause containing all the literals in the cube:

$$cube2clause(D) = \{L_1, L_2, \ldots, L_c\}$$

then *complement of a cube* D, in symbols $\overline{D}$, is given by:

$$\overline{D} = \overline{cube2clause(D)} = \{\overline{L_1}, \overline{L_2}, \ldots, \overline{L_c}\}$$

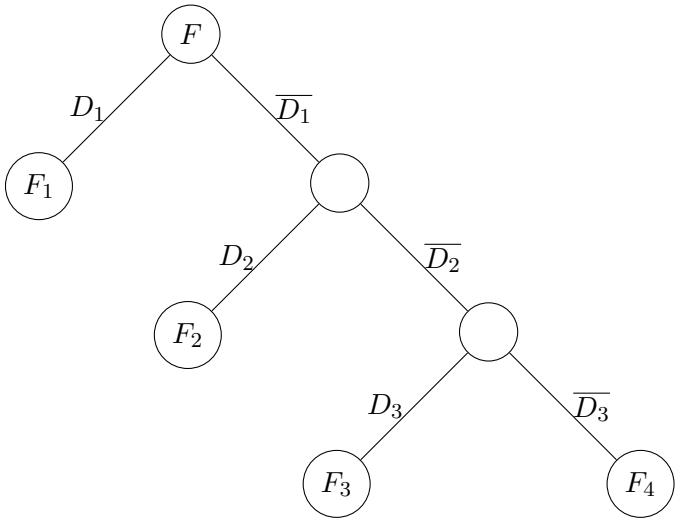


Figure 7: Example - Scattering Method for Creating Partitions

The search tree is biased in a sense that we can only split one node at each level of the tree, so w.l.o.g. we always split the right most node. Here is an example:

Example 4.6. Figure 7 shows scattering method for creating partitions. The right branch is a cube and the left branch is complement of the cube: a clause. You can see that we split only one of the nodes, at each level, in to two nodes using a complementary pair of cubes. Here are the partitions that are created by scattering method:

$$
\begin{aligned}
F_1 &= F \cup D_1 \\
F_2 &= F \cup \{\overline{D_1}\} \cup D_2 \\
F_3 &= F \cup \{\overline{D_1}\} \cup \{\overline{D_2}\} \cup D_3 \\
F_4 &= F \cup \{\overline{D_1}\} \cup \{\overline{D_2}\} \cup \{\overline{D_3}\}
\end{aligned}
$$

where D_1, D_2, D_3 are cubes, and $\overline{D_1}$, $\overline{D_2}$, $\overline{D_3}$ are the complements of cubes. For $1 \leq i \leq 4$, $F_i \setminus F$ are the *partitioning constraints*.

The number of splits we perform in scattering method depends on the number of partitions that we want to create, i.e. if we want to create 4 partitions, then we would perform three split operations. A point to note here is that scattering method provides a better control on the number of partitions than the simple method. The next important question is: how can we make these cubes? In other words, what should be the size of each cube i.e. the number of unit clauses in the cube? We describe one heuristic, given in [Hyv11], that is used to determine the size of each cube. Let k be the number of partitions we want to create, and let d_i be the size of i^{th} cube: the cube used to split the node at level $i-1$, then the idea is choose the values of d_i such that the partitions have the same expected solving time. We

represent expected solving time of i^{th} partition with T_i. To compute the values of d_i, we use the following equations [Hyv11]:

$$d_i = arg_{x \in \mathbb{N}} min |T_i - 2^{-x}|$$
$$\text{where} \qquad T_i = \frac{1}{k - i + 1}$$

We explain the heuristic, to determine the size of cubes for scattering method, with the help of the following example.

Example 4.7. Given F, we want to create four partitions: F_1, F_2, F_3, F_4. (see Figure 7)
Starting from the root node, we want to create four partition, so we divide the expected solving time by four, i.e. $T_1 = \frac{1}{4}$. Now the remaining partitions are three, so we divide the expected time by three, i.e. $T_2 = \frac{1}{3}$. Similarly, we get $T_3 = \frac{1}{2}$ and $T_4 = \frac{1}{1} = 1$. By find the values of d_1, d_2, d_3, d_4, each partition would look like the following:

$T_1 = \frac{1}{4}, \quad d_1 = 2, \quad F_1 = F \cup \{\{L_1\}, \{L_2\}\}$
$T_2 = \frac{1}{3}, \quad d_2 = 2, \quad F_2 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{L_3\}, \{L_4\}\}$
$T_3 = \frac{1}{2}, \quad d_3 = 1, \quad F_3 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{\overline{L_3}, \overline{L_4}\}\} \cup \{\{L_5\}\}$
$T_4 = 1, \quad d_4 = 0, \quad F_4 = F \cup \{\{\overline{L_1}, \overline{L_2}\}\} \cup \{\{\overline{L_3}, \overline{L_4}\}\} \cup \{\{\overline{L_5}\}\}$

4.2.2. Solving Partitions

Solving in search space partition is divided into two approaches [HJN10]: *plain partitioning* and *iterative partitioning.* To explain the difference between the two approaches, we first fix some notation. Some notation might look redundant now, but will be helpful later. We represent the search space partition solver by a partition tree, where each node ϕ is a tuple of the form $(Formula, Result, State)$, such that:

- $Formula$: is a CNF formula
- *Result* : can be *satisfiable*, or *unsatisfiable*, or unknown and represent them by symbols $\top$, $\bot$, ?, respectively
- *State* : can be *running* or stopped, represented by ▶, ■, respectively. *Running* means that the node is being solved by an incarnation and *stopped* means that the node is not being solved.

The root node $\phi(F, R, S)$ represents the given problem, where F is the given CNF formula that needs to be solved by search space partitioning solver. Each node $\phi'(F', R', S')$ can be expanded by partitioning the F, such that each partition of F is a child node of the node $\phi'(F', R', S')$. The key difference between the plain partitioning and the iterative partitioning is: only the leaf nodes are running state in the plain partitioning, while nodes other than the leaf nodes can also be in running state in iterative partitioning. We further explain the plain partitioning and the iterative partitioning with the following example.

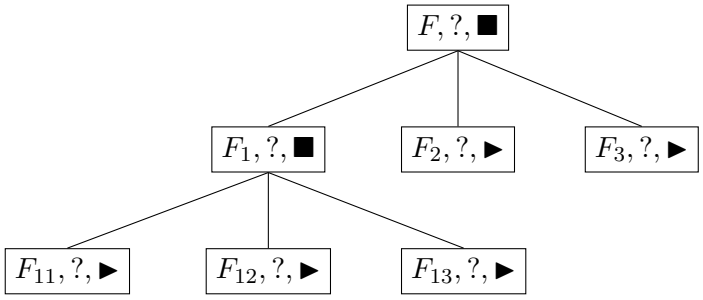


Figure 8: Example - Plain Partitioning

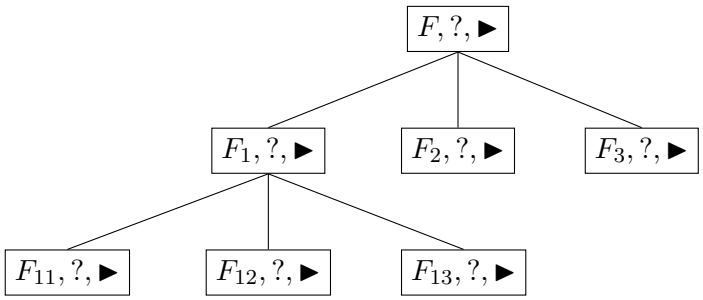


Figure 9: Example - Iterative Partitioning

Example 4.8. Figure 8 shows an example of plain partitioning and Figure 9 shows an example of iterative partitioning. In both figures, we partition the given CNF formula F (shown in the root node) into three partitions F_1, F_2, F_3, and we further partition F_1 into three partition F_{11}, F_{12}, F_{13}. The only difference between the two is in the state of the nodes. In Figure 8, only leaf nodes are in running state; note that we put in the nodes $\blacktriangleright$ symbol and $\blacksquare$ symbol for the state running and the state stopped.

As in practice, solver have limited number of resources to run incarnations, so in start, solver creates partitions according to the number of resources and then do *load balancing* when a resource becomes idle. In plain partitioning, load balancing can be done by *guiding path*. We will not cover the guiding path, you can refer to the original paper [ZBP+96]. For iterative partitioning, it is quite easy to do load balancing: solver just create new partitions in a breadth-first fashion of the partition tree or solver creates enough partitions in advance.

Now we discuss the difference between plain partitioning and iterative partitioning, on the basis of given CNF formula being satisfiable or unsatisfiable. In plain partitioning, solver can show the satisfiability of a given CNF formula by just showing the satisfiability of any leaf node $\phi'(F', R', S')$, i.e. the result R' of the leaf node is $\top$. For showing unsatisfiability of a given CNF formula in plain partitioning, solver needs to show that all leaf nodes are unsatisfiables, i.e. the result R' of each leaf node $\phi'(F', R', S')$ is $\bot$. In iterative partitioning, for proving satisfiability of a given

CNF formula, solver just need to show the satisfiability of any node $\phi'(F', R', S')$ in the partition tree; and for proving unsatisfiability of the given CNF formula, solver needs to show that there exist a node in every path from root to each leaf such that the result of the node is unsatisfiable.

To give the advantage of iterative partitioning over plain partitioning, we define two properties of a partitioning function: *ideal* and *void*.

Definition 4.9. Consider a CNF formula F with expected solving time T, and consider a partitioning function that created k partitions, with expected solving time T_i (for $1 \leq i \leq k$). The partitioning function is called **ideal**, if the estimated solving time of each partition is $T_i = \frac{T}{k}$. The partitioning function is called **void**, if the estimated solving of each partition is $T_i = T$.

In [HJN09], the author argues that obtaining ideal partitioning function is very difficult, but a partitioning function can be void more often, because modern SAT solvers use heuristics which may ignore certain variables. If the partitioning function chooses only these irrelevant variables for creating partitions, then the difficulty of the original CNF formula does not decrease. His result about the slow down with plain partitioning is given in the following proposition.

Proposition 4.10. *[HJN09] Given a unsatisfiable CNF formula F with expected solving time T, and plain partitioning approach with a void partition function, then the expected solving T_{plain} of the search space partitioning solver is greater or equal to the expected solving time of F, i.e. $T_{plain} \geq T$.*

Suppose that the partitioning function is void, then we can avoid the slow down introduced by plain partitioning by using iterative partitioning, because in iterative partitioning we are also solving the parents of the nodes; this result is due to [HJN10].

The iterative partitioning is originally used to solve the satisfiability problem in grid environment, given in [HJN10] and [Hyv11]. In grid environment, each resource is available for a certain time period, note that this time limit could be different for different resource and for that reason, the authors of [HJN10] and [Hyv11] introduce a limit on the solving time of each node in the partition tree, so when a time limit of a node is reached then the state of node is changed from running ▶ to stopped ■ and the result of the node remains unknown. An effort has been made in [HM12a] and [HM12b] to use the iterative partitioning for solving the satisfiability problem in a multi-core environment and the authors also use a time limit on nodes of the partition tree. My intuition is that a time limit on the nodes of the partition tree could be harmful and I will show in Section 5.1 that, in certain conditions, a time limit on nodes of the partition tree could introduce a slow down.

4.2.3. Sharing Information Among Partitions

A recent empirical study [KSMS11] shows that clause learning is the most important feature of modern SAT solvers. The same study also shows that the decision heuristic VSIDS is also an important feature of modern SAT solvers. Most of the current portfolio solvers share learnt clauses among the incarnations. This sharing of learnt clauses

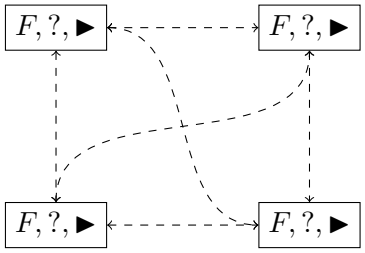


Figure 10: Example - Clause sharing in portfolio solvers

shows improvement in portfolio solvers, because a learnt clause can prune certain part of a search space of the incarnation. An important question in sharing learnt clauses is: which clauses are good for sharing? There is no successful answer to this question, but different portfolio solvers use different heuristics, e.g. ManySAT [HJS09b], winner of the *SAT competition 2009* in parallel SAT solver track, shares clauses of upto size eight, and PeneLoPe [AHJ+12], runner-up of the *SAT challenge 2012* in parallel SAT solver track, shares clauses of *LBD* value upto eight, and Plingeling [Bie12], winner of *SAT race 2010* and *SAT competition 2011* in parallel SAT solver track, shares only unit clauses and literal equivalences. Surprisingly, portfolio solver pfolioUZK [WvdGSP12], winner of *SAT challenge 2012* in parallel SAT solver track, does not share any clauses.

Example 4.11. Figure 10 shows an example of a portfolio solver that uses four incarnations. As each incarnation, shown by a node in the figure, solves the same given CNF formula, so each incarnation can share learnt clauses with every other incarnation. We have shown the clause sharing with dashed lines in the figure.

In the iterative partitioning, we can not share each learnt clause with every incarnation. This restriction is due to the partitioning constraints that can contribute to the learning of a clause, and so the clause learnt can not be a logical consequence of CNF formulas solved by other incarnations. The first approach for sharing clauses in the iterative partitioning is given in [HJN11], this clause sharing approach is called *flag-based learnt clause tagging*. The author gives the idea to tag each learnt clause with *safe* or unsafe, and share the safe learnt clauses. He defines unsafe clause inductively as: clauses belonging to partitioning constraints are *unsafe*, a learnt clause is *unsafe* it is obtained by resolution derivation involving one or more unsafe clauses. He defines safe clause as: a clause that is not unsafe is called *safe* clause. Sharing in the iterative partitioning has been further improved by *position-based clause tagging* [LM13]. The authors extend the idea of safe and unsafe clause in a sub-partition tree and sharing of learnt clause in a sub-partition tree if it is safe in that sub-partition tree. He attaches a position, in a partition tree, to each clause and calculates the position of a newly learnt clause, by taking the maximum of the position of the clause that is used in the resolution derivation for learning. Of course every clause learned by a node is semantic consequence of its decedents and can be shared with them, these clauses can be shared with position-based approach but not with flag-based approach.

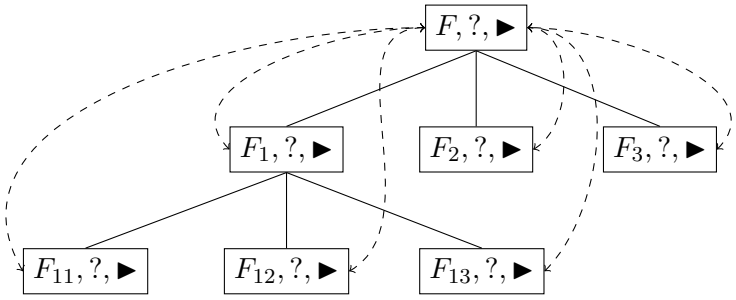


Figure 11: Example - Clause sharing with flag-based tagging in iterative partitioning

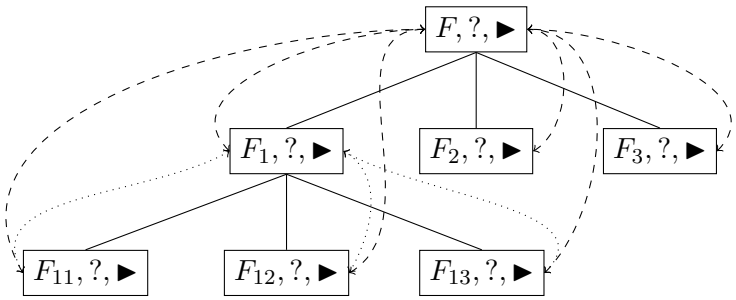


Figure 12: Example - Clause sharing with position-based tagging in iterative partitioning

Example 4.12. To elaborate the difference between flag-based and position-based clause tagging, we present an graphical example. Figure 11 shows an example of iterative partitioning that has incarnation distributed in three levels of the partition tree, and flag-based clause tagging. The flag-based clause tagging approach for clause sharing only shares clauses that are semantic consequence of the root node. We have shown this restriction by a directed dashed line in the figure. Of course, every clause learned by the root node can be shared. Figure 12 shows the position-based clause tagging approach in iterative partitioning. You can see that the position-based clause tagging approach is an extension of the flag-based approach, the dashed lines represent the flag-based approach. It shares the clauses in a sub-partition tree, if the clause is a semantic consequence of the nodes in that sub-partition tree. We represent the extended sharing with a dotted lines in the figure. You can see that the node $\phi''(F_{13}, ?, \blacktriangleright)$ learned a clause C that is semantic consequence of the node $\phi'(F_1, ?, \blacktriangleright)$, and this clause can be shared with $\phi'(F_1, ?, \blacktriangleright)$ and its decedents. You can think of the position-based approach as a flag-based approach at sub-partition tree level.

We can also share information about decision heuristic and polarity heuristic in the iterative partitioning, by propagating the information down the partition tree. In this way, a child node gets decision heuristic and polarity heuristic from its parent, when the child node starts its search.

4.3. Diversification vs Intensification

A search strategy in a modern SAT solver use the following components: decision heuristic, polarity heuristic, restart policy, and learning scheme. *Diversification vs intensification* is a trade-off made by the search strategy. **Intensification** refers to search strategies with goal to greedily improve the chances of finding a solution. **Diversification** strategies try to achieve a reasonable coverage of the search space. For further reading, you can refer to [HS04].

In the sense of sequential SAT solver, we see decision heuristic and polarity heuristic as a intensification strategy, while restart policy is a diversification strategy, and learning scheme is a mix of diversification and intensification. ManySAT [GHJS10], a portfolio solver, diversifies each of its incarnation by choosing different search strategy, and uses a master-slave model to intensify: master shares information about where he is in the search space, so that the slaves should look in the similar but not exactly the same search space. Best to our knowledge, there has been no effort with regard to the diversification vs intensification of a search space partitioning solver. We will discuss more about diversification and intensification in Section 5.

5. Improving Search Space Partitioning SAT Solver

In this section, we discuss the ideas for improving search space partitioning solver. We use SplitterLA [Irf12] as the base solver, which is an extended and improved version of the search space partitioning solver Splitter [HM12a] [HM12b]. Like its predecessor, SplitterLA is also a search space partitioning solver based on the iterative partitioning. The solver uses scattering method with lookahead techniques for creating partitions, and solves these partitions with CDCL solver MiniSat. We have discussed about search space partitioning solvers in Section 4.2, and discussed CDCL and lookahead in Section 3. This work also looks into the diversification and intensification for search space partitioning solver that uses iterative partitioning approach. We see different partitions created by search space partitioning solver as diversification strategy, and information sharing as intensification strategy. We discuss more about diversification vs intensification in Section 5.5.

We also present some test results, the outcome of our ideas, and these tests are run on a 16 core AMD Opteron 6274 CPUs with 2.2 GHz clock speed. We use a limit on memory, i.e. 8 GB, and a overall time limit of 7200 seconds per instance. We denote the time limit by *timeout*. The benchmark used for experiments is the set of instances from *SAT Competition 2009*, *SAT Competition 2011 unselected*, and *SAT Challenge 2012*; we denote this set by Ins. Let T_{S_i} be the time taken by solver S solving instance i, then the set $solvedIns(S)$ by a solver S, such that $solvedIns(S) \subseteq Ins$, is defined as:

$$solvedIns(S) = \{i \mid T_{S_i} < timeout \text{ and } i \in Ins\}$$

and set of unsolved instances $unsolvedIns(S)$ by solver S is:

$$unsolvedIns(S) = Ins \setminus solvedIns(S).$$

We use cactus plot, which has number of solved instances on x-axis and solving time on the y-axis. Let $Ins_{\leq t}(S)$ be the set of instances such that the time taken by solver S solving instance i is less or equal to t, i.e. $T_{S_i} \leq t$. We define cactus plot of solver S on instance set Ins as a set of data points (x, y) that are plotted on a two dimensional diagram:

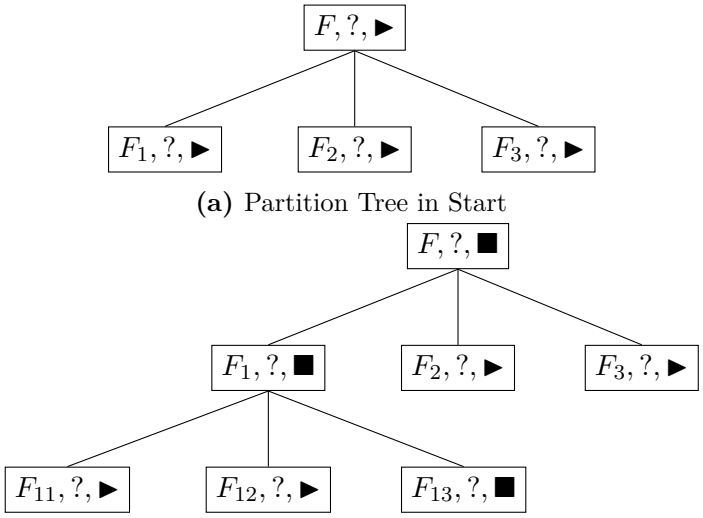
$$cactusPlot(S) = \{(x, y) \mid x = |Ins_{\leq y}(S)| \text{ and } y \in [0, timeout]\}.$$

We define cross plot of two solvers S' and S'' on instance set Ins as set of data points (x, y) that are plotted on a two dimensional axis:

$$crossPlot(S', S'') = \{(x, y) \mid i \in Ins \text{ and } x = T_{S'_i} \text{ and } y = T_{S''_i}\}$$

5.1. Solving Limit

SplitterLA uses a solving time limit for each node in the partition tree, proposed in [HJN10] for grid environment. Here we propose a different strategy, we do not put



(a) Partition Tree in Start

(b) Partition Tree after some time

Figure 13: Example - Iterative Partitioning with Limit

any limit on the solving time for each node. To give the intuition about our idea, we first define *ideal solving time limit:* a solving time limit is *ideal* if the given CNF formula is solved within that limit.

Given a unsatisfiable CNF formula that we solve with iterative partitioning having a void partition function, and if the solving time limit is not ideal then iterative partitioning slowly becomes plain partitioning. Figure 13a shows this scenario: iterative partitioning approach with four incarnations that can run in parallel, the CNF formula F, shown as root node, having three child nodes, all four nodes are solved in parallel. Root node $\phi(F, ?, \blacktriangleright)$ and its left most child $\phi(F_1, ?, \blacktriangleright)$ timeout with result unknown because the solving time limit is not ideal, and so their status become stopped, i.e. $\phi(F, ?, \blacksquare)$ and $\phi(F_1, ?, \blacksquare)$. As two resources become free, so the solver adds three child nodes to the node $\phi(F_1, ?, \blacksquare)$ by partitioning F_1, that are $\phi(F_{11}, ?, \blacktriangleright)$, $\phi(F_{12}, ?, \blacktriangleright)$, and $\phi(F_{13}, ?, \blacksquare)$ (two of the newly created nodes are running and one is stopped); see Figure 13b. Note that only leaf nodes are in running state, which is same as what plain partitioning approach does, so we can apply the Proposition 4.10 given on page 30, which says that plain partitioning approach with void partitioning function suffers a slowdown.

We test our hypothesis by running an experiment, in which we use the following two configurations:

Table 1: Statistics on 880 instances - *Limit* and *without Limit*

Configuration	Solved	SAT	UNSAT	Median Solving Time (sec)
Limit	489	256	233	1108.83
without Limit	**590**	**292**	**298**	**857.37**

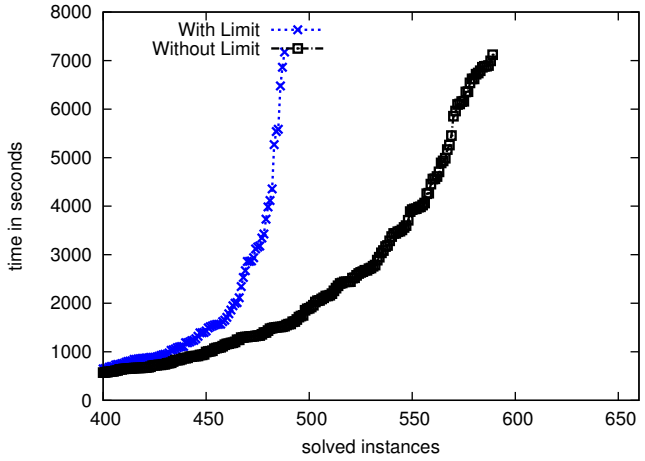


Figure 14: Cactus Plot - SplitterLA: Limit vs Without Limit

- *Limit:* SplitterLA with a solving time limit[1] for each node
- *without Limit:* SplitterLA without any solving time limit

According to our hypothesis, the configuration *without Limit* should perform better than *Limit*. Figure 14 shows a cactus plot of the two configurations, which we run over 880 instances with overall 7200 sec timeout limit using 8 cores and 8 GB memory. You can see that the configuration *without Limit* clearly outperforms (beyond our expectations) the configuration *Limit*, by solving 101 instances more, proves our hypothesis to be true. Table 1 shows the statistics of the configurations *Limit* and *without Limit* on 880 instances. *Limit* is able to solve 489 instances (256 satisfiable and 233 unsatisfiable) with a median solving time of 1108.83 sec, while *without Limit* is able to solve 590 instances (292 satisfiable and 298 unsatisfiable) with a median solving time of 857.37 sec.

5.2. Sequential Solver

The second big improvement we bring is by changing the SAT solver used to solve the partitions. SplitterLA uses MiniSat [Nik10] for solving partitions. We have build new search space partitioning solver with iterative partitioning, which uses Glucose [AS12a] to solve partitions. We call this new solver as SplitterGluLA. Glucose is the winner of *SAT challenge 2012* in the sequential SAT solver track, and

[1]Solving time limit is modeled as number of conflicts in SplitterLA

Table 2: Statistics on 880 instances - *SplitterLA* and *SplitterGluLA*

Configuration	Solved	SAT	UNSAT	Median Solving Time (sec)
SplitterLA	590	292	298	857.37
SplitterGluLA	644	292	352	548.04

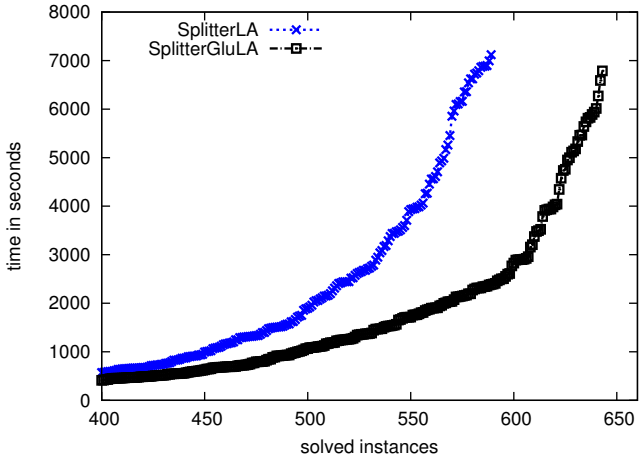


Figure 15: Cactus Plot - *SplitterLA* vs *SplitterGluLA*

is a modified version of MiniSat (ranked 9^{th} in *SAT challenge 2012* in the sequential SAT solver track), but with different learnt clauses cleaning policy and restart policy. Glucose uses LBD based learnt clauses cleaning policy and dynamic restart policy; we have discussed these policies in Section 3.1.

Figure 15 shows the cactus plot of the configuration *SplitterLA* and *SplitterGluLA*, where the configuration *SplitterLA* uses the solver SplitterLA and the configuration *SplitterGluLA* uses the solver SplitterGluLA. Both configurations do not have any limit on the solving time of a partition. You can see that *SplitterGluLA* solves faster and more instances than *SplitterLA*. Table 2 shows the statistics of the configurations. *SplitterGluLA* solves 644 instances (292 satisfiable and 352 unsatisfiable) with a solving time of 548.04 sec. *SplitterLA* solves less than *SplitterGluLA*, i.e. 590 instances (292 satisfiable and 298 unsatisfiable) with a median time of 857.37 sec. These results show that changing the SAT solver, for solving partitions, to a better SAT solver can improve the overall performance of the parallel SAT solver.

We use *SplitterGluLA* as the base configuration for further improvements, we put zero in the subscript, i.e. $SplitterGluLA_0$. As almost every solver participating in that the international competitions, uses a preprocessor, so for the rest of this work, we will use preprocessor SatElite [EB05] as well.

5.3. Instance Selection

For testing our ideas for improvements, we select *training set* of 118 SAT instances out of a set of 880 instances. We select a smaller subset for training so that we can perform test faster by using less resources, but we will still do the final evaluation on the set of 880 instances. We have made this selection after performing some experiments with the configurations $SplitterGluLA_0$, *PeneLoPe*, and *Plingeling*, where *PeneLoPe* runs the parallel solver PeneLoPe [AHJ+12] and *Plingeling* runs the parallel solver Plingeling [Bie12]. All the configurations use preprocessor SatElite. We selected the training set of 118 instances that are medium to hard instances: the selection

includes 72 unsatisfiable, 28 satisfiable instances, and 18 unsolved instances instances by all configurations. We use this training set to test our ideas, whether they bring improvements or not?

Table 3 shows the number of solved instances and other information by the configurations. *SplitterGluLA$_0$* solves 85 instances: 24 satisfiable instances and 61 unsatisfiable instances, with a median solving time of 1262.77 sec; while *PeneLoPe* solves 100 instances, of which 28 are satisfiable and 72 are unsatisfiable instances, and it has a median solving time of 303.33 sec. *Plingeling* solves 90 instances that include 28 satisfiable and 62 unsatisfiable instances, with a median solving time of 1276.06 sec. Although we have brought two major improvements, solving limit and the partition solver, still *SplitterGluLA$_0$* needs more improvements to catch up with *PeneLoPe*, but *SplitterGluLA$_0$* is not too far from *Plingeling*. You should note that *PeneLoPe* and *Plingeling* uses clauses sharing and other techniques, which we look into, later in this section.

5.4. Modifications in Creating Partitions

We want to improve the average partitioning time of the solver, as it effects the overall performance of the solver (there could be a scenario that resources are waiting for the creation of partitions), in that case the solver may not utilize all the given resources efficiently. Average partitioning time should be small enough to utilize parallel resources more efficiently, so our aim to lower this time without loosing overall performance of the solver.

5.4.1. Pre-selection Heuristic

As we discussed in Section 3.2, that pre-selecting small number of variables for *lookaheadDecide* can improve overall performance of lookahead SAT solver; in our case, we can improve the performance of creating partitions, i.e. less time to create partitions. *SplitterGluLA$_0$* uses a static method for choosing the size of pre-selection variables: top 10% of the free variables, with a maximum cutoff of 2048 variables. We now add the adaptive ranking method (see page 16). We have tested four different configuration for adaptive ranking:

1. $\alpha = 5$ and $\beta = 7$, this is default configuration of lookahead SAT solver MARCH [HvM06].
2. $\alpha = 10$ and $\beta = 7$.

Table 3: Statistics on 118 selected instances - *SplitterGluLA$_0$*, *PeneLoPe*, *Plineling*

Configuration	Solved	SAT	UNSAT	Median Solving Time (sec)	Avg. Partitioning Time (sec)
SplitterGluLA$_0$	85	24	61	1262.77	31.31
PeneLoPe	**100**	**28**	**72**	**303.33**	-
Plineling	90	28	62	1276.06	-

3. $\alpha = 20$ and $\beta = 7$.

4. $\alpha = 50$ and $\beta = 7$.

We have found the fourth configuration to perform slightly better than the other three configurations.

5.4.2. Double Lookahead

Lookahead SAT solvers get benefits from using double lookahead in *lookaheadDecide*, but we have seen in [Irf12] that use of double lookahead in partitions creation degrades the performs of the solver. Here, we revisit the idea of double lookahead. We propose to use different pre-selection heuristic for double lookahead than the one used for *lookaheadDecide*. As the purpose of double lookahead is to find failed literal which was not detected by applying single lookahead. In other words, consider applying lookahead on literal L_1 and the lookahead is not successful, then the goal of double lookahead is to find a variable A such that lookahead on $L_2 = A$ is successful and lookahead on $\overline{L_2}$ is also successful (see Section 3.2). So, we change the pre-selection heuristic for double lookahead by choosing variables whose both polarities are present in large number of the newly created binary clauses (binary clauses that have been created by applying lookahead on L_1).

Consider a CNF formula F and an interpretation J, we apply lookahead on literal L_1, i.e. $lookahead(F :: J, \dot{L_1})$, such that lookahead on L_1 is unsuccessful, then we can pre-select variables for double lookahead with the following heuristic:

$$F :: J, \dot{L_1} \rightsquigarrow^*_{\text{UNIT}} F :: J, \dot{L_1}, P$$

$$newBin(F :: J, \dot{L_1}, P) = \{C \text{ , such that } C \in F|_{J,\dot{L_1},P} \text{ and } C \notin F \text{ and } |C| = 2\}$$

$$\#freq_2(L_2) = |\{C \text{ , such that } L_2 \in C \text{ and } C \in newBin(F :: J, \dot{L_1}, P)\}|$$

$$preselectDLAscore(A) = \#freq_2(A) * \#freq_2(\neg A)$$

where $newBin(F :: J, \dot{L_1}, P)$ represents the newly created binary clause in F by interpretation $J, \dot{L_1}, P$, $\#freq_2(L_2)$ is the frequency of the literal L_2 present in $newBin(F :: J, \dot{L_1}, P)$, and $preselectDLAscore(A)$ is the heuristic score for variable A. We use adaptive ranking method (see page 16) to choose the pre-selection variable set for double lookahead.

5.4.3. Tabu Scattering

In our experiments, we have observed that scattering method creates partitions such that there are common variables among the cubes used for splitting nodes. Recall the scattering method is defined as, a biased search tree that splits a node on a complementary pair of a *cubes* D_1 and $\overline{D_1}$ (see Section 4.2.1 for details). We now define *tabu scattering method* as an extension of scattering method, by putting a restriction that a variable used in one cube, for splitting a node, must not be used

in the cubes for splitting other nodes. Consider a CNF formula F, we create four partitions F_1, F_2, F_3, F_4 using three cubes D_1, D_2, D_3:

$$\begin{aligned} F_1 &= F \cup D_1 \\ F_2 &= F \cup \overline{D_1} \cup D_2 \\ F_3 &= F \cup \overline{D_1} \cup \overline{D_2} \cup D_3 \\ F_4 &= F \cup \overline{D_1} \cup \overline{D_2} \cup \overline{D_3} \end{aligned}$$

The restriction by tabu scattering is given by:

$$atom(D_1) \cap atom(D_2) \cap atom(D_3) = \emptyset$$

Using tabu scattering method, we get two benefits: first is that we diversify the search more, second is that we lower the probability of the partitioning function being a void function.

5.4.4. Pure Literals

The paper [DP60], on which most of the modern SAT solvers are based on, gives the pure literal rule: a literal which exists only either in positive polarity or in negative polarity in the CNF formula. Modern SAT solvers usually do not use the pure literal rule, because their data structures focus on detecting cheaply the literals which are present in unit clauses, and detecting pure literals with these data structures is computationally expensive. In *SplitterGluLA$_0$*, we use preselection heuristic, which has to anyhow go through the CNF formula, for *lookaheadDecide*; we can add the PURE rule in during the score calculation for preselection heuristic, without introducing computational overhead. The detected pure literal can be added as partitioning constraint for a partition.

We define pure literal rule in the DPLL ARS.

Definition 5.1. The **pure literal** rule can be described using the set R_{DPLL} from DPLL ARS.

$$F :: J \;\rightsquigarrow_{\text{PURE}}\; F :: J, L^l \qquad \text{, iff } \nexists C \in F|_J \text{ such that } \overline{L} \in C \text{ , and } l = level(J)$$

The pure literal appends the pure literal to the interpretation. Here is an example:

Example 5.2. Consider the following CNF formula:

$$F = \{\{\neg 1, 2\}, \{\neg 1, \neg 2, 3\}, \{\neg 1, \neg 3, 4\}, \{1, 3, 6\}, \{\neg 1, 4, \neg 5\}, \{1, \neg 6\}, \{4, 5, 6\}\}$$

then, the literal 4 is pure, because the literal $\neg 4$ does not exist in any clause of F, and we can apply the pure literal rule.

$$\begin{array}{ll} & F :: () \\ \rightsquigarrow_{\text{PURE}} & F :: (4^0) \quad F|_{(4^0)} = \{\{\neg 1, 2\}, \{\neg 1, \neg 2, 3\}, \{1, 3, 6\}, \{1, \neg 6\}\} \end{array}$$

5.4.5. Constraint Resolvents

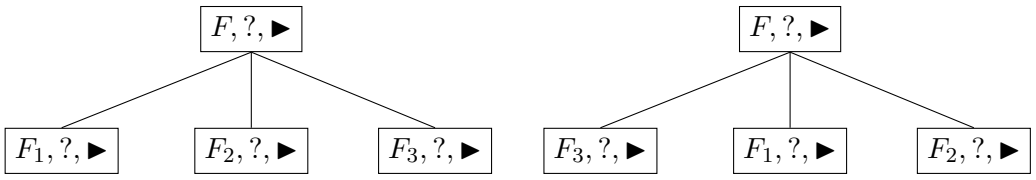
We perform local reasoning while performing *lookaheadDecide* (see page 15), i.e. add local learnt clauses to the CNF formula. *SplitterGluLA*$_0$ adds failed literals, necessary assignments, and equivalent literals. We improve *SplitterGluLA*$_0$ by adding constraint resolvents as local learnt clauses during *lookaheadDecide*. Constraint resolvents are loosely defined as the local learnt clauses which are useful: useful in the sense that these local learnt clauses help in detecting more failed literals and necessary assignments. We use a simple heuristic to add constraint resolvents, i.e. we add all indirect implications $\{\overline{L_1}, L_2\}$ found after performing lookahead on the literal L_1, as local learning, if the the reason clause for L_2 is not a binary clause. In symbols, we represent this heuristic as *conRes*:

$$\begin{aligned} \mathit{conRes}(F :: J, \dot{L_1}) = \{\{\overline{L_1}, L_2\} \mid & C \in F \text{ and } C \text{ is reason for } L_2 \in \mathit{iup}(F :: J, \dot{L_1}) \\ & \text{and } |C| > 2\} \end{aligned}$$

5.4.6. Sorting Children

We have observed in experiments that scattering method does not always create partitions that have equal difficulty level in terms of solving time. Due to this difference, consider a scenario that the solver has some resources free, so it creates partitions of some running unsolved node $\phi(F_i, ?, \blacktriangleright)$ in the partition tree, but it may happen that $\phi(F_i, ?, \blacktriangleright)$ is very close to find the result $\bot$ and thus the solver may waste resources on the newly created partitions. We propose a solution to decrease the chance of this scenario to happen, by sorting the child nodes, of each parent node, in decreasing order of difficulty level; this way the solver will create partitions of more difficult node first than the less difficult nodes. We predict the difficulty level of a node by a simple heuristic that counts the number of propagated literals: higher the number of starting propagated literals means lower difficulty level. Number of starting propagated literals of a CNF formula F is given by $\#\mathit{propagation}(F)$, which is defined as:

$$\#\mathit{propagation}(F) = |s(P)|$$



(a) Scattering without sorting **(b)** Example - Scattering with sorting

Figure 16: Sorting Children

$$\text{such that } F :: (\ ) \rightsquigarrow^{*}_{\text{UNIT}} F :: P$$

where $s(P)$ is the set of propagated literals. We explain more with the help of an example.

Example 5.3. Consider the solver creates three partitions of a F, with scattering method, we represent the partition tree shown in Figure 16a. Suppose that:

$$\#propagation(F_3) < \#propagation(F_1) < \#propagation(F_2)$$

so we predict that partition F_2 is has the lowest difficulty level, then comes the partition F_1, and the hardest among all the partitions is F_3; so with sort option, the solver changes the position of the partitions in the partition tree, and is shown in Figure 16b.

5.4.7. Results

We perform an experiment using two solver configuration: *SplitterGluLA$_0$* and *SplitterGluLA$_1$*. For creating partitions, both solver use scattering method with lookahead decision heuristic, use recursive weighted heuristic as pre-selection heuristic, add equivalent literals as local learnt clauses, detect failed literals and necessary assignments, add failed literals and necessary assignments to partitioning constraints. The differences between these two solvers are: *SplitterGluLA$_0$* uses 10% with maximum cutoff of 2048 variables for choosing the number of pre-selection variables, while *SplitterGluLA$_1$* uses adaptive ranking with lower bound equal to 50 and failed literals importance factor equal to 7. Apart from these options, *SplitterGluLA$_1$* detects and adds pure literals to partitioning constraints, adds constraint resolvents as local learnt clause, performs double lookahead, and imposes tabu restriction on scattering. We use test set of 118 selected instances for performing the experiment; using 8 cores, 8 GB memory and 7200 timeout per instance. Figure 17 shows a cactus plot of these two solvers. *SplitterGluLA$_1$* performs better by solving 87 instances than *SplitterGluLA$_0$*, which solves 85 instances. Table 4 shows more detailed information of the experiment. Median Solving time is slightly better for *SplitterGluLA$_1$*, i.e. 1255.42 sec as compared to *SplitterGluLA$_0$* with 1262.77 sec. Surprisingly, *SplitterGluLA$_1$* uses about a factor of 6 less time to create partitions per node, compared to *SplitterGluLA$_0$*. *SplitterGluLA$_1$* also improves the CPU ratio, that is the measure of the utilization of resources. In the next subsection, we discuss ideas to further improve *SplitterGluLA$_1$*.

Table 4: Statistics on 118 instances - Improvements in Partition Creation

Configuration	Solved	Median Solving Time (sec)	Avg. Partitioning Time (sec)	CPU Ratio
SplitterGluLA$_0$	85	1262.77 sec	31.31 sec	6.34
SplitterGluLA$_1$	**87**	**1255.42** sec	**4.71** sec	**6.45**

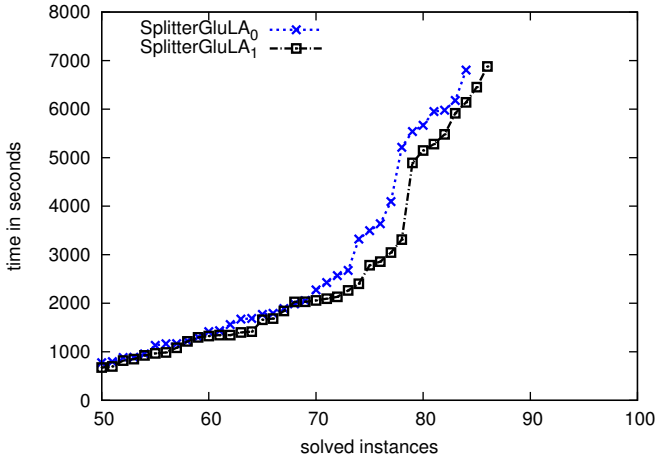


Figure 17: Cactus Plot - Improvements in Creating Partitions

5.5. Diversification vs Intensification in Solving Partitions

Now we take ideas about diversifying and intensifying the search of parallel solver, from portfolio solvers. Creating partitions is itself a diversification process, we discuss more ideas to further diversify and intensify the search of *SplitterGluLA*$_1$.

5.5.1. Sharing VSIDS and Progress Saving

We intensify the search by sharing information. First thing we look into is sharing VSIDS and progress saving heuristics. Portfolio solvers do not share this information, because all incarnations start their search at the same time; but in case of our solver, we have a tree structure (partition tree) that we can exploit, and also nodes in the partition tree do not start search at the same time. So, sharing heuristic information like VSIDS and progress saving from parent to child nodes, could help the child nodes (given that the parent node starts search before child node). When *SplitterGluLA*$_1$ starts solving, the root node and the nodes at the partition tree level one start at almost the same time, because the solver creates partitions of a node equal to the number of resources. The nodes at partition tree level greater than one are usually created after some time, so we initialize the new child nodes with VSIDS and progress saving information of their parent, because the child node searches in the sub-search space of its parent and whatever is learned by parent node can help the child node as well. We get improvement in the performance of *SplitterGluLA*$_1$ by sharing VSIDS heuristic scores and progress saving values of polarity, from nodes at partition level one or greater, to their child nodes; we discuss the results in Section 5.5.8.

5.5.2. Sharing Learnt Clauses

Parallel SAT solvers share learnt clauses among incarnations to intensify the search. We look into this idea and use position based learnt clause sharing in *SplitterGluLA*$_1$; we have discussed position based clause sharing in Section 4.2.3. We use the best

known configuration for position based clause from [LM13], that shares clauses of LBD score less or equal to two. We present this configuration as a function $shareClause(C)$, which return $\top$ if the clause C can be shared or return $\bot$ otherwise; in symbols we present this as:

$$shareClause(C) = \begin{cases} \top \text{ , if } lbd(C) \leq 2 \\ \bot \text{ , otherwise} \end{cases}$$

As expected, we get improvement by adding clause sharing option in $SplitterGluLA_1$ (see Section 5.5.8).

5.5.3. Sharing Top Level Units

By top level units, we mean the literals present at decision level zero in the interpretation. Inspired from PLINGELING that shares top level units among its incarnations, we can share top level units the same way as we share learnt clauses, by treating top level units as unit clauses. We have conducted experiments to share top level units, and we see a slowdown in overall performance of the solver. There might be several reasons, but the ones we could think of are:

- sharing them increase the communication overhead (as they are large in number)
- other nodes could have found the top level unit themselves because of the shared learnt clauses, sharing of VSIDS and progress saving information

We have not deeply investigated about the reason for slowdown, and leave this for future work. For the time being, we do not use this option in our solver.

5.5.4. Different Restarts

Portfolio solvers like MANYSAT and PENELOPE, use different restart policies for each incarnation, to diversify the search. Inspired by this idea, we diversify by using different restart policy parameters. First we classify the nodes in partition tree into three categories:

1. *Root node:* the node at root of the partition tree.
2. *Leaf node:* the node which does not has any child node.
3. *Middle node:* the node which is neither a root node nor a leaf node.

According to these node categories, we apply different restart policies.

- Root node uses the default restart policy of GLUCOSE, i.e. $X = 50$ and $K = 0.8$.
- Leaf node uses $X = 75$ and $K = 0.8$.
- Parent node uses $X = 75$ and $K = 0.7$.

Recall that X is the size of bounded queue and K is the magic constant (see page 12), that are used in dynamic restart policy by GLUCOSE. We have selected the values of X and K based on experiments and the data provided in [AS12b].

5.5.5. Randomizing Polarity

Although sequential solvers make use progress saving for choosing the polarity, but in start the solver does not has this information; so usually sequential solvers choose negative polarity for a decision variable when no progress saving has been done for that variable. *PeneLoPe* uses this fact by directing some incarnations to select random polarity of a decision variable if the variable does not have any saved progress (last used polarity); this small change brings some diversification in *PeneLoPe* because different incarnations may choose the same decision variable but may take different polarity (different direction of search). We use the same idea in the following way:

- Root node chooses the default negative polarity for a decision variable if no progress saving information is available.
- Nodes at partition level one, use the random if no progress saving information is available.

Note that this idea does not conflict with VSIDS and progress sharing, because we only introduce this idea for the nodes which start almost at the same time, i.e. root node and the nodes at partition tree level one.

5.5.6. Different Learnt Clauses Cleaning

To diversify, PeneLoPe also uses different interval for performing learnt clause cleaning (time between two learnt clause cleanings), for different incarnations. The purpose is that some incarnations keep learnt clauses for longer time, while others for shorter. We introduce this idea in our solver, according to the node category. We give different cleaning intervals to root node, middle node, and leaf node, in such a way that root node has the shortest cleaning interval, middle node has longer cleaning interval but shorter than leaf node. Let Int_{root}, Int_{middle}, Int_{leaf} be the cleaning intervals of the nodes root, parent, and leaf, respectively. Then we have the following relationship:

$$Int_{root} < Int_{middle} < Int_{leaf}$$

Note that a leaf node changes its cleaning policy dynamically when it become a middle node (when it has child nodes).

5.5.7. Only Child Scenario

During our experiments we have observed, on some instances, that the height of the partition tree grows equal to the number of parallel resources (number of cores in our case). This means that there is only one unsolved node at each partition level of the partition tree. On a smaller scale, there could be only one unsolved node at some partition level. For that reason, we call this scenario the *only child scenario*. Here is an example.

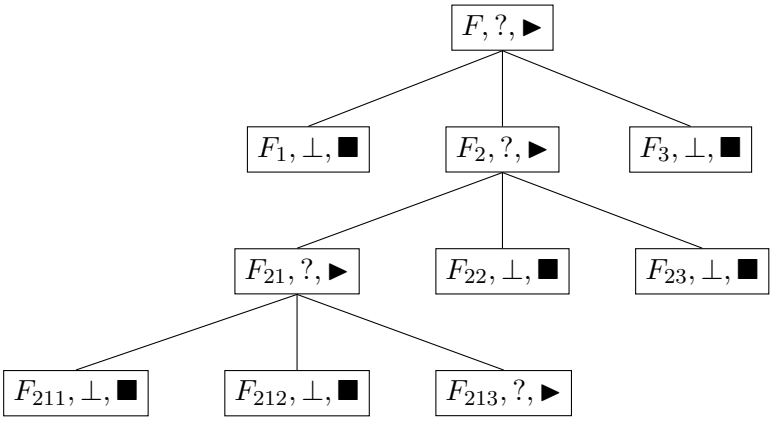


Figure 18: Example - Only Child Scenario

Example 5.4. Figure 18 shows an extreme case of only child scenario: solver with four available resources. You can see that only one node is unsolved at each level of partition tree, i.e. the nodes solving the partitions F, F_2, F_{21}, F_{213} are unsolved and running.

Consider that only child scenario happens at some level of the partition tree, then there are two cases:

- parent node is looking in the search space which has been solved by one of its child.
- parent node is looking in the search space where its unsolved children are looking.

In either case, we have the risk of doing redundant work. We propose a approach to get out of this scenario by reintroducing solving limit in a node that has only one unsolved child, as we can say with a high probability that the partitioning function is not void (as all the partitions are solved except one). To be on safe side, we do not introduce this limit in the root node if it has only one unsolved child, so that we are not slower than the sequential solver. The introduced limit grows with level of the partition tree.

Remark. We do not know if the only child scenario is good or bad, as portfolio solvers also have the risk of doing redundant work and still they perform well. We suggest to further investigate this only child scenario as a future work. One possible direction could be to simulate portfolio approach[2] in iterative partitioning.

5.5.8. Results

For testing the combined effect of the ideas presented in this subsection, we conduct an experiment using three solver configurations: *SplitterGluLA*$_1$, *SplitterGluLA*$_2$,

[2]Idea from the discussion with Davide Lanti

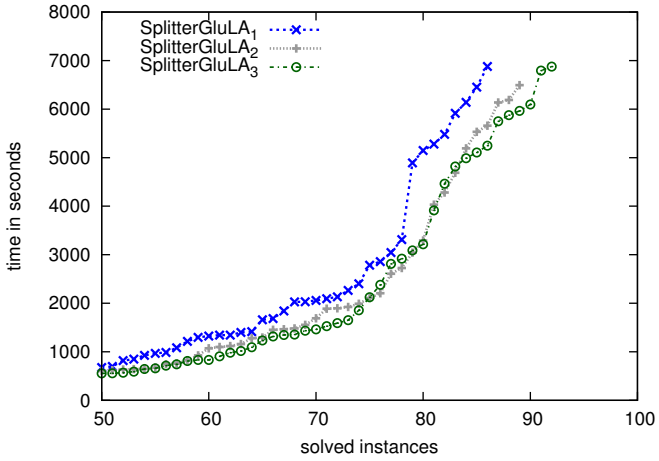


Figure 19: Cactus Plot - Improvements by Diversification and Intensification

and *SplitterGluLA*$_3$. The experiment is run over the test set of 118 instances, using 8 cores with solving time limit of 7200 sec, memory limit of 8 GB. *SplitterGluLA*$_1$ is the improved configuration from Section 5.4. *SplitterGluLA*$_2$ and *SplitterGluLA*$_3$ are extensions of *SplitterGluLA*$_1$. *SplitterGluLA*$_2$ and *SplitterGluLA*$_3$ use the intensification and diversification techniques, i.e. sharing VSIDS and progress saving, sharing learnt clauses, different restarts, randomizing polarity, different learnt clauses cleaning; we have discussed these techniques in this subsection. The only difference between *SplitterGluLA*$_2$ and *SplitterGluLA*$_3$ is the only child scenario: former ignores the only child scenario, while latter considers the only child scenario by adding solving time limit to the node with only one unsolved child.

Figure 19 shows the cactus plot of the three configurations, and as you can see that we get improvements by applying diversification and intensification techniques (*SplitterGluLA*$_2$ and *SplitterGluLA*$_3$): the number of solved instances has increased and the median solving time has decreased. Among these configurations, *SplitterGluLA*$_3$ solves the most number of instances, i.e. 93 instances (27 satisfiable and 66 unsatisfiable), see Table 5. It solves more than *SplitterGluLA*$_2$ which solves 90 instances (25 satisfiable and 65 unsatisfiable). Also the median solving time of *SplitterGluLA*$_3$ (820.9 sec) is better than the median solving time of *SplitterGluLA*$_2$ (875.48 sec). These results motivate the importance of only child scenario, and we think further research in this direction will bring some good results.

Table 5: Statistics on 118 instances - Improvements by Diversification and Intensification

Configuration	Solved	SAT	UNSAT	Median Solving Time (sec)	CPU Ratio
SplitterGluLA$_1$	87	26	61	1255.42	**6.45**
SplitterGluLA$_2$	90	25	65	875.48	6.14
SplitterGluLA$_3$	**93**	**27**	**66**	**820.9**	6.42

6. Evaluation

In this section we discuss the result of our experiments, which include, finding a good configuration, analysis of speed and efficiency, test for scalability, and comparison with state-of-the-art portfolio solvers. In the end we also show the improvements contributed through this work.

The hardware setup, that is used in our experiments, has 16 core AMD Opteron 6274 CPUs with 2.2 GHz clock speed. We run our experiments over an instance set Ins of 880 instances (set of instances from *SAT Competition 2009*, *SAT Competition 2011 unselected*, and *SAT Challenge 2012*).

6.1. Good Configuration

To find a good configuration for SplitterGluLA, we test four configurations. These configurations are selected from the experiments performed in Section 5.4 and Section 5.5. Following are the details of the configurations:

1. *wOCS:* (with only child scenario) limiting the solving time of a parent node if it has only one unsolved child.

2. *wOCSDelayClean:* (with only child scenario + different learnt clauses cleaning) limiting the solving time of a parent node if it has only one unsolved child and using different learnt clauses cleaning intervals according to the category of the partition tree node.

3. *wOCSDelayCleanRndPol:* (with only child scenario + different clauses cleaning + randomizing polarity) limiting solving time of a parent node if has only one unsolved child, and using different learnt clauses cleaning intervals according to the category of the partition tree node, and using random polarity in the nodes at partition level one, for deciding a literal when no progress saving information is available.

4. *woOCS:* (without only child scenario) does nothing if a parent node has only one unsolved child.

All of the above mentioned four configurations use the same options for creating partitions (*SplitterGluLA*$_1$ from Section 5.4); all configurations also use sharing VSIDS and progress saving option, sharing learnt clauses option, different restarts option (for

Table 6: Statistics on 880 instances - Final Evaluation

Configuration	Solved	SAT	UNSAT	Median	CPU ratio	Score
wOCS	680	299	381	179.48	**6.05**	−33
wOCSDelayClean	**682**	**300**	**382**	175.10	6.02	**14**
wOCSDelayCleanRndPol	676	298	378	171.99	5.75	13
woOCS	676	296	380	**163.77**	5.97	6

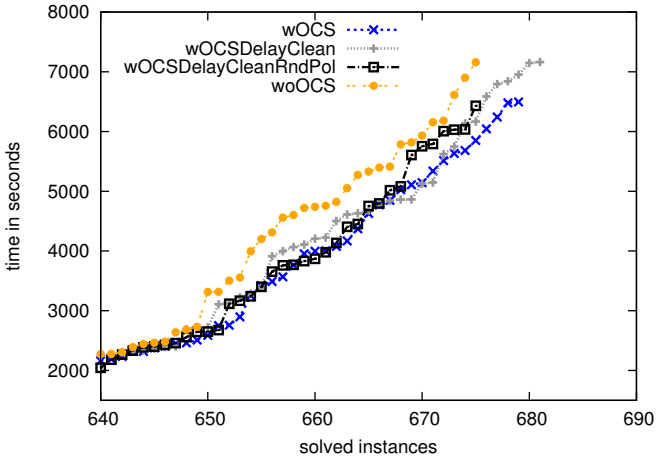


Figure 20: Cactus Plot - Final Evaluation

details about these options, see Section 5.5). All configurations can use 8 core. The time limit is set to 7200 sec (2 hours) and memory limit is set to 8 GB.

Figure 20 shows a cactus plot of the four configurations (we have discussed the definition of a cactus plot on page 34). You can see that the configurations which consider the only child scenario (*wOCS*, *wOCSDelayClean*, *wOCSDelayCleanRndPol*), perform better than the configuration which does not consider the only child scenario (*woOCS*), on hard instances. Although the median time of *woOCS* is the lowest among all the configurations, but it solves less number of instances compared to *wOCS* and *wOCSDelayClean*, see Table 6. Since we are particularly interested in solving hard instances, *wOCSDelayClean* seems to do a good job, as it solves the highest number of instances, i.e. 682 instances (300 satisfiable and 382 unsatisfiable). A careful ranking system for solvers [VG11], that considers ties (difference of solving time between two solvers is less than 300 sec) as noise, is shown in the last column of the table, which places *wOCSDelayClean* as first by giving 14 score, and 13 score, 6 score, and −33 score is given to *wOCSDelayCleanRndPol*, *woOCS*, and *wOCS*, respectively. Interesting score comparison is between *wOCS* and *woOCS*: the former gets low score but solves more instances, whereas later gets high score but solves less instances. This comparison shows the importance of the only child scenario that it should not be ignored. *wOCS* makes a trade-off over the speed (on easy instance) with solving hard instances. The important question which arise here is: can this trade-off be minimized, such that the speed (on easy instances) does not become slow and still solves more number of hard instances? This question motivates to further look into the only child scenario.

6.2. Comparison with Sequential Solver

SplitterGluLA uses Glucose to solve partitions, we compare the performance of SplitterGluLA with sequential Glucose. The configuration *Sequential* represents a sequential solver Glucose, and *Parallel-8core* (using 8 cores) represents the config-

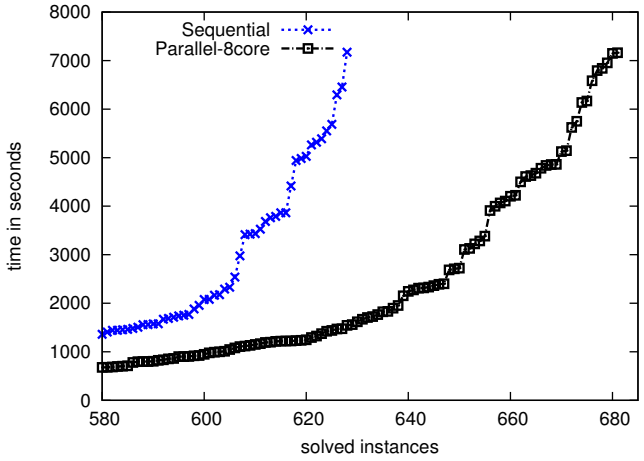


Figure 21: Cactus Plot - Sequential vs Parallel

Table 7: Statistics on 880 instances - Sequential vs Parallel

Configuration	Solved	SAT	UNSAT	Median	CPU ratio	Score
Sequential	629	272	357	289.28	-	−96
Parallel-8	**682**	**300**	**382**	**175.10**	6.02	**96**

uration *wOCSDelayClean* of SplitterGluLA from our experiments, in Section 6.1. Figure 21 shows a cactus plot of the configurations *Sequential* and *Parallel-8core* and it is clear that *Parallel-8core* is solving more instances and also solving instances faster. *Sequential* solves 629 instances (272 satisfiable and 357 unsatisfiable), while *Parallel-8core* solves 682 instances (300 satisfiable and 382 unsatisfiable). Table 7 shows the statistics of the experiment. The median solving time is better for *Parallel-8core* (175.10 sec) than *Sequential* (289.28 sec). The score for *Parallel-8core* is 96, while the score for *Sequential* is much lower, i.e. −96; this shows improvement of using parallel approach over sequential.

Table 8 shows the speedup analysis of the configuration *Parallel-8*. We have discussed speedup on page 22. We observe:

- occasional super linear speedup on few instances (19 satisfiable instances and 4 unsatisfiable instances), where *Parallel-8core* is 8 time faster than *Sequential*,
- linear speedup on 421 instances (187 satisfiable instances and 234 unsatisfiable instances), where *Parallel-8core* is faster than *Sequential*,

Table 8: Statistics on 880 instances - Speedup analysis for 8 cores

	Super Linear Speedup		Linear Speedup		Slowdown	
Configuration	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Parallel-8	19	4	187	234	92	150

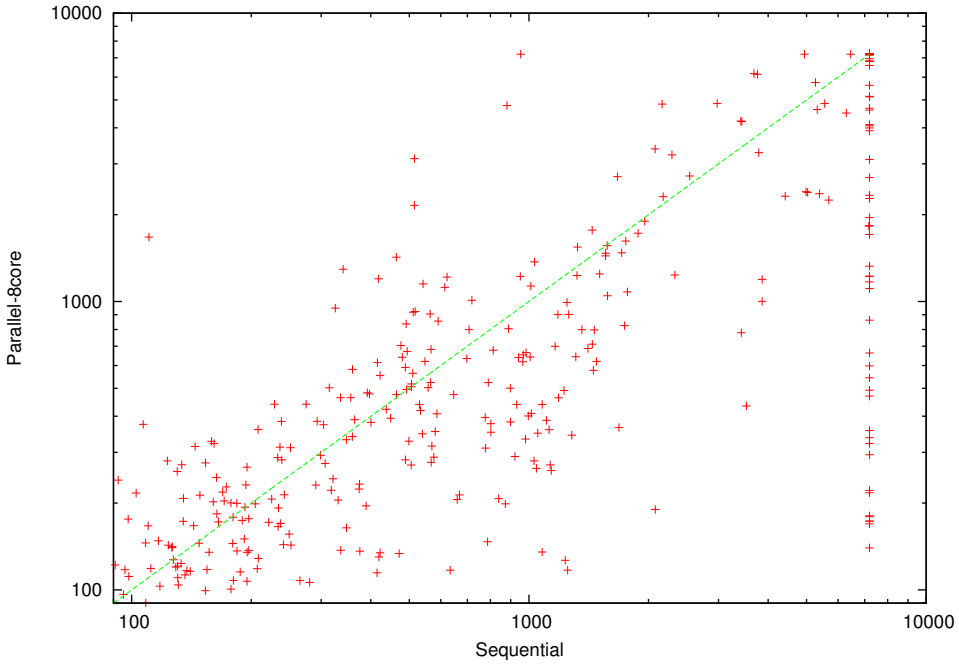


Figure 22: Cross Plot - *Sequential* on y axis and *Parallel-8core* on y axis

- slowdown in 242 instances (92 satisfiable and 150 unsatisfiable) where *Parallel-8core* is slower than *Sequential*.

A cross plot of *Parallel-8core* vs *Sequential* in Figure 22 shows the impact of this slowdown (cross plot is defined on page 34); x-axis shows the solving time of *Sequential* and y-axis shows the solving time of *Parallel-8core*. The figure shows that most of the slower solved instances by *Parallel-8* compared to *Sequential*, do not have a big difference in the solving time, except for only few instances.

Overall the number of instances where speedup is observed (linear or super linear), is twice the number of instances where slowdown. We still suggest to further investigate the reasons for slowdown on some instances and look into implementation of SplitterGluLA; one direction could be sharing of some bad clauses (bad in a sense that these clauses mislead the search) with the root node in the partition tree.

6.3. Scalability Test

A scalable parallel SAT solver means that the performance of the solver increases with the increase in the number of resources. To check if our approach for parallel SAT solving is scalable, we use the configurations *wOCSDelayClean* and *woOCS* of SplitterGluLA (from Section 6.1). *wOCSDelayClean-4*, *wOCSDelayClean-8*, and *wOCSDelayClean-16* represent the configuration *wOCSDelayClean* using 4-cores, 8-cores, and 16-cores, respectively. Same notation is applied to *woOCS*. We also use PeneLoPe using 4-core, 8-core, and 16-cores. We choose PeneLoPe because it is claimed [AHJ+12] to scale better than the other portfolio solvers like Plin-

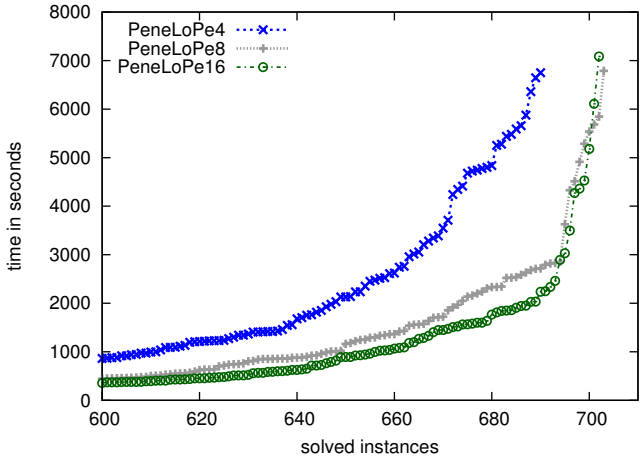


Figure 23: Cactus Plot - *PeneLope* 4-core, 8-core, 16-core

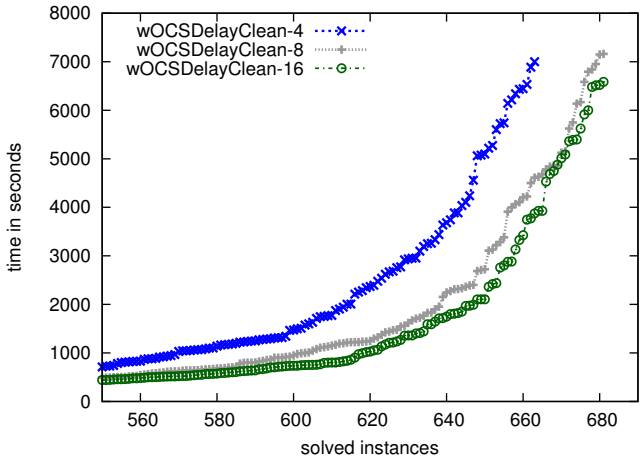


Figure 24: Cactus Plot - *wOCSDelayClean* 4-core, 8-core, 16-core

GELING [Bie12], CRYTOMINISAT [Soo10], MANYSAT [HS09], and PPFOLIO [Rou12]. *PeneLoPe-4*, *PeneLoPe-8*, and *PeneLoPe-16* represent the configuration of PENELOPE using 4-core, 8-core, and 16-core, respectively.

We use *penalized average runtime-10 (par10)* to measure the performance of each configuration. Let S be a solver, Ins be the instance set, $timeout$ be the solving time limit, T_{S_i} be the time taken by S to solve instance $i \in Ins$, $solvedIns(S)$ be the set of solved instances by S, and $unsolvedIns(S)$ be the set of unsolved instances by S (see page 34 for more details), then par10 of S (we denote it by $par10(S)$) is defined as:

$$par10(S) = 10 * |unsolvedIns(S)| * timeout + \sum_{i \in solvedIns(S)} T_{S_i}$$

We say that an approach to parallel SAT solving is scalable if it par10 score decreases with the increases in number of resources.

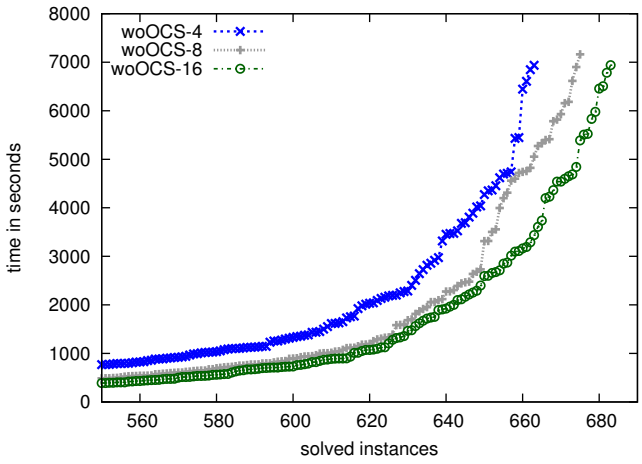


Figure 25: Cactus Plot - *woOCS* 4-core, 8-core, 16-core

Table 9: Statistics on 880 instances - Scalability Analysis

Configuration	Solved	SAT	UNSAT	Median	par10
PeneLoPe-4	691	298	393	140.86	13,913,759
PeneLoPe-8	704	304	400	89.39	**12,887,177**
PeneLoPe-16	703	302	401	**83.42**	12,916,426
wOCSDelayClean-4	664	295	369	212.92	15,906,877
wOCSDelayClean-8	682	300	382	175.10	14,580,060
wOCSDelayClean-16	682	299	383	**149.8**	**14,531,761**
woOCS-4	664	294	370	207.44	15,876,623
woOCS-8	676	296	380	163.77	14,984,328
woOCS-16	684	299	385	**150.18**	**14,397,826**

Figure 23 shows the cactus plot of *PeneLoPe-4*, *PeneLoPe-8*, and *PeneLoPe-16*. It is clear from the figure that *PeneLoPe-8* solves 13 more instances than *PeneLoPe-4*, but *PeneLoPe-16* solves one less instance than *PeneLoPe-8*. Figure 24 shows the cactus plot of *wOCSDelayClean-4*, *wOCSDelayClean-8*, and *wOCSDelayClean-16*; this plot shows that *wOCSDelayClean-8* solves 18 more instances than *wOCSDelayClean-4*, but *wOCSDelayClean-16* solves same number of instances compared to *wOCSDelayClean-8*. Figure 25 shows the cactus plot of *woOCS-4*, *woOCS-8*, and *woOCS-16*. This plot shows a bit better picture (solving more instances by adding more resources) than previous plots shown in Figure 23 and Figure 25. *woOCS-8* solves 12 more instances than *woOCS-4*, and *woOCS-16* solves 8 more instances compared to *wOCSDelayClean-8*.

Table 9 shows the statistics of 4-core, 8-core, and 16-core configurations of *PeneLope*, *wOCSDelayClean*, and *woOCS*. All the configuration shows decrease in median solving time with the increase in number of cores. *PeneLoPe-8* has lower par10 score than *PeneLoPe-4*, but *PeneLoPe-16* has higher par10 score than *PeneLoPe-8*, so it seems that PeneLoPe does not scale from 8-core to 16-core. par10 score for *wOCSDelayClean* and *woOCS* decreases with the increase in the number of cores. The decrease in par10 score is more significant in *woOCS* than *wOCSDelayClean*. This suggests that SplitterGluLA scales better than portfolio solvers (as it scales slightly better than PeneLoPe, and PeneLoPe scales better than other portfolio solvers [AHJ+12]). We believe that a better implementation of SplitterGluLA could further improve its scalability.

6.4. Comparison with Other Parallel SAT Solvers

We compare our solver with state-of-the-art portfolio solvers: PeneLoPe (winner of *SAT race 2010* and *SAT competition 2011*, in parallel track) and Plingeling (runner-up of *SAT challenge 2012* in parallel track). We use the instance set *Ins* of 880 instances for experiment and three configuration:

1. *PeneLope* using PeneLoPe,
2. *Plingeling* using Plingeling,
3. *SplitterGluLA* using the configuration *wOCSDelayClean* of SplitterGluLA from Section 6.1.

Each configuration runs using 8-cores with 8 GB memory limit and 7200 sec time limit. Figure 26 shows a cactus plot of the solver *SplitterGluLA*, *PeneLoPe*, and *Plingeling*. It is quite evident that *SplitterGluLA* performs better *Plingeling*: both in terms of

Table 10: Statistics on 880 instance - Parallel Solvers Comparison

Configuration	Solved	SAT	UNSAT	Median	CPU ratio	Score
SplitterGluLA	682	300	382	175.10	6.02	26
Plingeling	672	296	376	442.28	6.38	−291
PeneLoPe	**704**	**304**	**400**	**89.39**	**6.90**	**265**

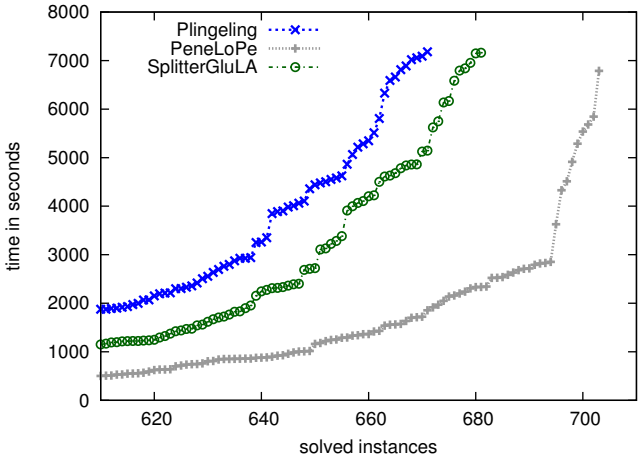


Figure 26: Cactus Plot - Parallel Solvers Comparison

number of solved instances and the time to solve an instance. *SplitterGluLA* solves less instances and is almost twice slower than *PeneLoPe*. Table 10 shows the statistics of the experiment: *SplitterGluLA* solves 682 instances (300 satisfiable and 382 unsatisfiable) with median solving time of 175.10 sec, *Plingeling* solves 672 instances (296 satisfiable and 376 unsatisfiable) with median solving time of 442.28 sec, and *PeneLoPe* solves 704 instances (304 satisfiable and 400 unsatisfiable) with median solving time of 89.39 sec. The big difference among these configurations is the number of unsatisfiable solved instances, *SplitterGluLA* solves 18 less unsatisfiable instances than *PeneLoPe*. This suggests to focus more on unsatisfiable instances. Another important observation is the CPU ratio, *SplitterGluLA* has the lowest value, i.e. 6.02. In ideal case the CPU ratio should be equal to the number of cores, but due to the communication overhead between the incarnation, the CPU ratio in practice is lower than the number of cores. This means *SplitterGluLA* is wasting almost 2 out of 8 cores. CPU ratio for *Plingeling* is 6.38 which is slightly better than *SplitterGluLA*. *PeneLoPE* has the best CPU ratio, i.e. 6.90. SplitterGluLA follows object oriented approach (with virtual functions) in implementation, and since SplitterGluLA is based on iterative partitioning, which has a chance of having some cores idle (cores are waiting for the creation of new partitions). Another reason for low CPU ratio of SplitterGluLA and high CPU ratio of PeneLoPe is the time limit on preprocessing phase. There are few instances in the *Ins* that SatElite takes more than 1000 sec to preprocess them, it means that only one core is utilized till preprocessing is done. SplitterGluLA uses SatElite without any time limit on the preprocessing phase, while PeneLoPe uses SatElite with a time limit of 300 sec (PeneLoPe kills SatElite if the time limit exceeds and continues solving the instance without preprocessing). This motivates to use preprocessor that can provide the CNF formula within certain time limit. These might be the reasons of low CPU ratio, but still we will look into the implementation of SplitterGluLA to improve the CPU ratio.

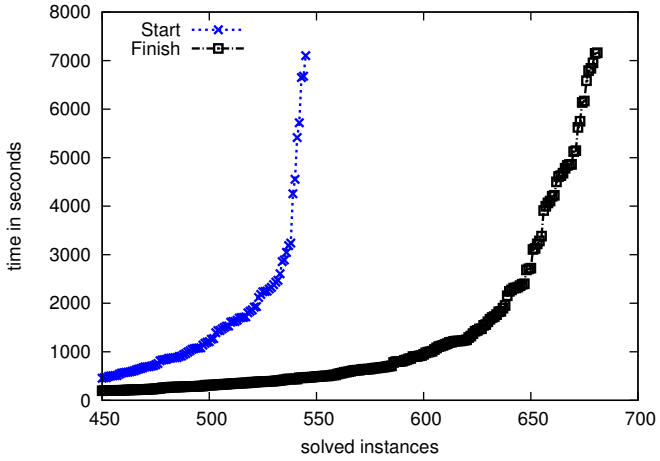


Figure 27: Cactus Plot - Improvements of this work

These result show that SplitterGluLA can compete with the current portfolio solvers, and encourage to further research iterative partitioning approach.

6.5. Improvements of this work

We show the progress of this work in Figure 27: a cactus plot of two configurations *Start* and *Finish*. *Start* uses SplitterLA (starting point of this work) and *Finish* uses SplitterGluLA (finish point of this work). It is clear from the figure that we have brought huge improvements in this work, as *Finish* is able to solves 136 instances more than *Start*. Also *Finish* has a better median solving time of 175.10 sec than *Start* with median solving time of 362.14 sec.

7. Conclusion

The outcome of this work is a solver called SplitterGluLA, which is a search space partitioning SAT solver based on iterative partitioning approach. SplitterGluLA is a huge step forward w.r.t. the previous work SplitterLA [Irf12]. We have shown that SplitterGluLA is comparable in performance with competitive parallel SAT solvers, like Plingeling (winner of *SAT competition 2011*-parallel track) and PeneLoPe (runner-up of *SAT challenge 2012*-parallel track). Based on par10 measure, we have also shown that SplitterGluLA seems to scale slightly better than the best known scalable portfolio solver, PeneLoPe. The improvements of this work are:

- removing limits on the solving time in iterative partitioning, as we have shown that the limit degrades the overall performance of the solver,
- using state-of-the-art sequential SAT solver Glucose as the solver underlying the parallel solver,
- creating partitions with lookahead [Irf12] faster, by reducing dramatically the partition creation time, we decrease the chance of starvation in the solver (scenario when cores are idle and waiting for new partitions to be created),
- applying the techniques of diversification and intensification in the solver, by taking some ideas from the state-of-the-art portfolio SAT solvers.

For future work, we believe that exploring the only child scenario (see page 45) could bring encouraging results, e.g. the idea of simulating portfolio in iterative partitioning seems interesting, as the clauses from the only unsolved child can be shared with its parent without any restriction. We have seen that the performance of SplitterGluLA decreases with the sharing of top level units among the partition node, but this sharing shows good results in Plingeling. Investigating why is the case could also be a possible future work direction. We have applied preprocessing to the given CNF formula once before the solver start search; we think that applying fast preprocessing on each node in the partition tree (created by iterative partitioning) would further improve the performance of the solver.

Moreover, PeneLoPe uses a new freeze-unfreeze [AHJ+12] method to manage the learnt clauses in each incarnation; adding this method to the solver, that solves partitions in SplitterGluLA, could bring further improvements.

A. List of Figures

B. List of Tables

References

[ABL+10] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In Inês Lynce and Ralf Treinen, editors, *LoCoCo*, volume 29 of *EPTCS*, pages 11–22, 2010.

[AF10] Dimitrios Athanasiou and Marco Alvarez Fernandez. Recursive weight heuristic for random k-SAT. Technical report, Delft University of Technology, 2010.

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

[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.

[AS12a] 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.

[AS12b] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *18th International Conference on Principles and Practice of Constraint Programming (CP'12)*, pages 118–126, oct 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 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.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Bie12] Armin Biere. Lingeling and friends entering the sat challenge 2012. In Balint et al. [BBD+12], pages 33–34. ISBN 978-952-10-8106-4.

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

[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.

[CS12] Alessandro Cimatti and Roberto Sebastiani, editors. *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*. Springer, 2012.

[DD04] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-SAT formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 305–306. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24605-3_36.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(7):1165–1178, July 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, Berlin, Heidelberg, 2005. Springer-Verlag.

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[GHJS10] Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 252–265, Berlin, Heidelberg, 2010. Springer-Verlag.

[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.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '02, pages 142–, Washington, DC, USA, 2002. IEEE Computer Society.

[Goe10] Asvin Goel. A column generation heuristic for the general vehicle routing problem. In *Proceedings of the 4th international conference on Learning and intelligent optimization*, LION'10, pages 1–9, Berlin, Heidelberg, 2010. Springer-Verlag.

[GSC97] Carla Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search, 1997.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1-2):67–100, February 2000.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[HDvZvM05] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: implementing additional reasoning into an efficient look-ahead SAT solver. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, SAT'04, pages 345–359, Berlin, Heidelberg, 2005. Springer-Verlag.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *J. Artif. Int. Res.*, 36(1):267–306, September 2009.

[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.

[HJN09] Antti E. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning search spaces of a randomized search. In *Proceedings of the XIth International Conference of the Italian Association for Artificial Intelligence Reggio Emilia on Emergent Perspectives in Artificial Intelligence*, AI*IA '09:, pages 243–252, Berlin, Heidelberg, 2009. Springer-Verlag.

[HJN10] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer-Verlag.

[HJN11] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *Proceedings of the 17th international conference on Principles and practice of constraint programming*, CP'11, pages 385–399, Berlin, Heidelberg, 2011. Springer-Verlag.

[HJS08] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: Solver description. In *In SAT race*, 2008.

[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 *Accepted for HVC 2011*, 2012. Accepted for HVC 2011.

[HM12a] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing scalable parallel SAT solvers. In Cimatti and Sebastiani [CS12], pages 214–227.

[HM12b] Antti Eero Johannes Hyvärinen and Norbert Manthey. Splitter – a scalable parallel SAT solver based on iterative partitioning. In Balint et al. [BBD+12], page 62. ISBN 978-952-10-8106-4.

[HMN+11] Steffen Hölldobler, Norbert Manthey, Hau Van Nguyen, Julian Stecklina, and Peter Steinke. Modern parallel sat-solvers. Technical Report Technical Report 2011-06, TU Dresden, Dresden, Germany, 2011.

[HS04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[HS09] Youssef Hamadi and Lakhdar Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[HvM06] Marijn J.H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, mar 2006.

[HvM07] Marijn J.H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.

[HvM09] Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. Volume 185 of Biere et al. [BHvMW09], February 2009.

[Hyv11] Antti Eero Johannes Hyvärinen. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Helsinki, Finland, 2011. Doctoral Dissertations 118/2011.

[Irf12] Ahmed Irfan. Search space partitioning with lookahead. Technical Report Technical Report 2012-04, TU Dresden, Dresden, Germany, 2012.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI '92, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[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.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371. Morgan Kaufmann, 1997.

[Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Inf. Process. Lett.*, 71(2):75–80, July 1999.

[Li03] Chu-Min Li. Equivalent literal propagation in the dll procedure. *Discrete Appl. Math.*, 130(2):251–276, August 2003.

[LM13] Davide Lanti and Norbert Manthey. Sharing information in parallel search with search space partitioning. Technical report, Dresen University of Technology, 2013.

[LMS06] Inês Lynce and João Marques-Silva. SAT in bioinformatics: Making the case with haplotype inference. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 136–141. Springer, 2006.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[Man10] Norbert Manthey. Improving SAT Solvers Using Stat-of-the-Art Techniques. Master's thesis, TU Dresden, Germany, 2010.

[MdWH10] Sid Mijnders, Boris de Wilde, and Marijn J. H. Heule. Symbiosis of search and heuristics for random 3-SAT. In David Mitchell and Eugenia Ternovska, editors, *Proceedings of the Third International Workshop on Logic and Search (LaSh 2010)*, 2010.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.

[MSLM09] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. Volume 185 of Biere et al. [BHvMW09], February 2009.

[Nik10] Niklas Sörensson. MINISAT 2.2 and MINISAT++ 1.1. `http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf`, 2010.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[Rou12] Olivier Roussel. Description of ppfolio 2012. In Balint et al. [BBD+12], page 46. ISBN 978-952-10-8106-4.

[Rya04] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.

[sat] The international SAT competitions. `http://www.satcompetition.org/`.

[Sin06] Daniel Singer. *Parallel Resolution of the Satisfiability Problem: A Survey*. Wiley Interscience, October 2006.

[SK93] Bart Selman and Henry A. Kautz. An empirical study of greedy local search for satisfiability testing. In Richard Fikes and Wendy G. Lehnert, editors, *AAAI*, pages 46–51. AAAI Press / The MIT Press, 1993.

[Soo10] Mate Soos. Cryptominisat 2.5.0. In *SAT Race competitive event booklet*, July 2010.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarcks's proof procedure for propositional logic. In Ganesh Gopalakrishnan and

Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 1998.

[VG11] Allen Van Gelder. 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, pages 317–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[WvdGSP12] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, and Stefan Porschen. pfolioUZK: Solver description, 2012. ISBN 978-952-10-8106-4.

[ZBP+96] Hantao Zhang, Maria Paola Bonacina, Maria Paola, Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

Declaration

Hereby I certify that this report has been written by me. Any help that I have received in my research work has been acknowledged. Additionally, I certify that I have not used any auxiliary sources and literature except for those cited in this report.

Dresden, 03 April 2013

Ahmed Irfan