Philosophæ doctor thesis

Hoessen Benoît

Solving the Boolean Satisfiability problem using the parallel paradigm

Jury composition:

PhD director	AUDEMARD Gilles
	Professor at Université d'Artois
PhD co-director	JABBOUR Saïd
	Assistant Professor at Université d'Artois
PhD co-director	PIETTE Cédric
	Assistant Professor at Université d'Artois
Examiner	SIMON Laurent
	Professor at University of Bordeaux
Examiner	DEQUEN Gilles
	Professor at University of Picardie Jules Vernes
	KATSIRELOS George
	Chargé de recherche at Institut national de la recherche agronomique, Toulouse

Abstract
This thesis presents different technique to solve the Boolean satisfiability problem using parallel and distributed architectures. In order to provide a complete explanation, a careful presentation of the CDCL algorithm is made, followed by the *state of the art* in this domain. Once presented, two propositions are made. The first one is an improvement on a portfolio algorithm, allowing to exchange more data without loosing efficiency. The second is a complete library with its API allowing to easily create distributed SAT solver.
Keywords: *SAT*, *parallelism*, *distributed*, *solver*, *logic*

Résumé

Cette thèse présente différentes techniques permettant de résoudre le problème de satisfaction de formule booléenes utilisant le parallélisme et du calcul distribué. Dans le but de fournir une explication la plus complète possible, une présentation détaillée de l'algorithme CDCL est effectuée, suivi d'un état de l'art. De ce point de départ, deux pistes sont explorées. La première est une amélioration d'un algorithme de type portfolio, permettant d'échanger plus d'informations sans perte d'efficacité. La seconde est une bibliothèque de fonctions avec son interface de programmation permettant de créer facilement des solveurs SAT distribués.
Mot-clés: *SAT*, *parallelisme*, *calcul distributé*, *solveur*, *logique*

Acknowledgments

Even tough there is only one name as the author of this thesis, this work is the result of a team effort. First and foremost, I would like to thank my supervisors: Gilles AUDEMARD, *Saïd* JABBOUR *and Cédric* PIETTE. *They provided me the opportunity to work on a exciting subject, and countless advices. I would like to thank the examiners Laurent* SIMON *and Gilles* DEQUEN , *George* KATSIRELOS *and the president of the jury TODO for accepting their respective tasks.*

I would also like to thank the members of the CRIL. Through the different informal discussions, they provided me their point of view, allowing me to change mine and be more confident about some choices I made.

A thesis in not only a scientific journey but also a complete experience. I would like to thanks all people who I encountered in this journey. Fred, Frédéric, Olivier, Yakoub and Daniel for the numerous and passionate debates we had at lunch. Nicolas and Vincent for the pleasant time in the office. My former colleagues at Amadeus Marianne, Florent, Fred, Colin, Fabien, Matthieu, Benoit, Paul, Thomas and every other for their encouragement to undertake such challenge. Mathilde Kurzawa for proofreading this thesis. I would also like to thanks my former teachers at the University of Liège, and professor Pascal GRIBOMONT *in particular. Last but not least, Claire and my family, for their comprehension and support.*

Foreword

Logic is an old subject studied in phylosophy, mathematics and computer science. The fact that it was already studied by Aristotle (384–322 BCE) shows that logic is not very new. Nevertheless, logic is still widely used to this day, and maybe even more than before. Indeed, it is at the heart of computer science through the boolean logic and its implication: central processing units, memory chips, ...

Nowadays, logic finds yet another usage in computer science through automatic validation of code, circuitry. Such validations are performed by theorem prover that found their use way beyond the computer science fields. It is now possible to find such prover in biology, mathematics, economics, and many more. Those provers usually work by finding an answer to a given combinatorial problem. One of the specificity of combinatorial problems is, according to the current state of our knowledge, that different solutions have to be enumerated. And such enumeration can be quite time consuming. One of those problem is the Boolean satisfaction problem (or *SAT* for short). This problem is very interesting for multiple reasons. First, any combinatorial problem can be expressed as a Boolean satisfaction problem. Second, the problem by itself can be easily explained and rely on logic operation. And as said earlier, logic is very well studied, therefore it provides information on how to solve the problem. However, even with the help of powerful rules, enumeration is still needed. This means that powerful algorithm are needed in order to reduce the time spent for the computation.

By searching to obtain an answer the fastest way possible, some scientists have focused on the parallel paradigm. Such paradigm allows to make multiple operations concurrently. This can be achieve through the use of multiple computers or by multiple 'cores' in a single computer. And luckily, those architectures are now the norm for personal computers and high-end cellphones. This grants access to such hardware for many scientific groups.

More than hardware, software is also needed for the parallel paradigm. Of course, the parallel paradigm is not the key to tackle the problem but it may provide new concepts that could help the research on the subject. Moreover, with the rise of practical problem being solved through SAT, being able to provide answers faster can be extremely helpful for the users of SAT technologies.

There are two family of algorithms to solve the SAT problem in parallel. The first one is the portfolio approach. Different competiting prover are run concurrently. The philosophy behind this algorithm is that the number of potential solutions is too high. By providing different provers, those will different enumeration order and therefore, one of them might find the solution faster. The second approach is the divide and conquer family of algorithms. The idea for those is that the number of potential solutions is too high. Therefore we can not accept that the same potential solution is checked twice or more. To do this, the formula is divided into two sub-formulæ or more, and each sub-formulæ is given to a standalone SAT solver.

The aim of this thesis is to propose two new approaches for solving the SAT problem using the parallel paradigm. The first one, `PeneLoPe`, is a member of the portfolio family of algorithms Through the use and combination of some of state of the art technique, we were able to create an award winning parallel SAT solver. The second, `dolius`, is a platform dedicated to distributed SAT solving. It aims to facilitate the creation of distributed solvers with a lot of flexibility on the division of the work.

This manuscript is composed of two groups of chapters. The first one explains the SAT problem, its origins and unique position in terms of computability (Chapter 1), followed by some explanations of algorithms to solve it (Chapter 2). The second presents the contributions of this thesis: an award-winning approach for clause exchange within a portfolio (Chapter 4) and a framework providing a simple interface to create distributed solvers (Chapter 5). Finally, a conclusion and some perspective are presented (Chapter 6), followed by a french résumé (Chapter 7) and the exhaustive list of contribution of this thesis (Chapter 8).

Contents

CHAPTER 1

Introduction

A man provided with paper,
pencil, and rubber, and subject to
strict discipline, is in effect a
universal machine.
—Alan Turing

In this chapter, we introduce some concepts needed to understand this thesis: logic and computer science. Logic is a key element in computer science as electronics are made of transistors gathered together to make complex logic gates that ultimately define our modern computers.

1.1 Logic

Logic is the science of reasoning and the expression of such. Therefore, by establishing correct rules of deduction, we can reason about the validity of arguments. Initially a complete philosophical question, knowing the validity of argument became extremely relevant in science as a proof is an argument. Logic defines proof systems to help validating such arguments.

1.1.1 Boolean algebra

Boolean algebra is an algebra not defined on numbers like most well-known algebra, but on the values *true* (hereby represented by the symbol $\top$) and *false* (hereby represented by the symbol $\bot$). Variables will be denoted as lowercase letters: a, b, c and may be indexed for the ease of some explanations. Their domains is the set of two elements: $\{\top, \bot\}$.

Functions of degree n in Boolean algebra can be expressed as $\mathcal{F}(a_1, \ldots, a_n) : \{\top, \bot\}^n \rightarrow \{\top, \bot\}$ and are also called predicates of degree n. There are exactly 2^{2^n} possible predicates of degree n. When $n = 1$, those 4 predicates are shown in Table 1.1: identity, $\top$, $\bot$ and the negation $\neg$.

When $n = 2$, there are 16 predicates and some of them will be studied further: the disjunction operator $\vee$, the conjunction operator $\wedge$, the exclusive-or $\otimes$, the *nand* $\overline{\wedge}$, the nor $\veebar$, the equivalence $\equiv$ and the implication $\Rightarrow$. Those are depicted in Table 1.2.

a	a	$\top$	$\bot$	$\neg a$
$\top$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\top$

Table 1.1: Every possible unary operator

a	b	$a \vee b$	$a \wedge b$	$a \otimes b$	$a \barwedge b$	$a \veebar b$	$a \equiv b$	$a \Rightarrow b$
$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$

Table 1.2: Some common binary Boolean operators

Usually, when n gets higher we define the different predicates by providing the corresponding formulæ. A formula Σ is a string of symbols respecting the following grammar:

variables: formula ::= a | b | c | ...

negation: formula ::= $\neg$ *formula*

binary operators: bop ::= $\wedge$| $\vee$ | $\otimes$ | $\barwedge$ | $\veebar$ | $\equiv$ | $\Rightarrow$

concatenation: formula ::= (*formula bop formula*)

We define $\Pi(\Sigma)$ as the set of variables occurring in Σ, $\Pi^+(\Sigma)$ the set of variables occurring positively (a, b, c) and $\Pi^-(\Sigma)$ those occurring negatively $(\neg a, \neg b, \neg c)$ in Σ.

Example 1.1: Formula Let us define the formula Σ using 3 variables a, b, c as $\Sigma = (a \vee (b \equiv \neg c))$. This is a valid formula as it can be derived from the grammar as shown with the derivation tree depicted in Figure 1.1

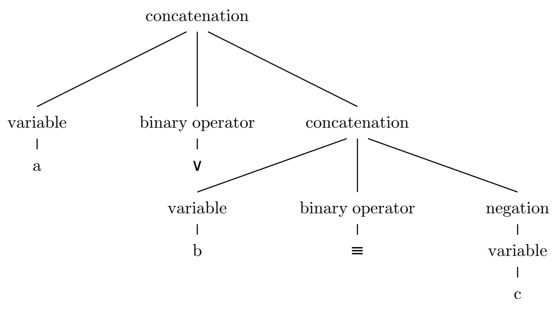


Figure 1.1: Derivation tree for the formula defined in Example 1.1

From this formula, we have $\Pi(\Sigma) = \{a, b, c\}, \Pi^+(\Sigma) = \{a, b\}$ and $\Pi^-(\Sigma) = \{c\}$.

End of Example 1.1

For a given formula, we may associate an assignation function $\mathcal{A} : \pi \subseteq \Pi(\Sigma) \rightarrow \{\top, \bot\}$ that maps a truth value to each variable of the formula. An assignation function is said to be complete when $\pi = \Pi(\Sigma)$ and incomplete when $\pi \subset \Pi(\Sigma)$.

Given a complete assignation function $\mathcal{A}$, we can deduce the truth value of a formula Σ by replacing the variables by their values. If Σ is evaluated to $\top$, $\mathcal{A}$ is said to be a model of Σ. When a model exists for a formula Σ, Σ is said to be consistent (or satisfiable). If no such model exists Σ is said to be inconsistent (or unsatisfiable). If every possible assignation function $\mathcal{A}$ is a model of Σ, Σ is said to be a tautology. From those definitions, it is easy to see that Σ is a tautology if and only if $\neg\Sigma$ is unsatisfiable.

Σ is said to be a logical consequence of Σ' if every model of Σ' is also a model of Σ. In this case, this will be written as $\Sigma' \models \Sigma$. If $\Sigma' \models \bot$, Σ' is unsatisfiable. If $\models \Sigma$, Σ is a tautology.

Example 1.2: consistency, tautology and inconsistent formulaes Let Σ_1 be the formula $\neg a \vee (b \vee c)$. We can define a complete assignation function $\mathcal{A}_1(a) = \top, \mathcal{A}_1(b) = \bot, \mathcal{A}_1(c) = \bot$. It is easy to see that the assignation function $\mathcal{A}_1$ is not a model of Σ_1 as the formula Σ_1 becomes $\bot \vee (\bot \vee \bot) = \bot \vee \bot = \bot$. As we found one assignation that is not a model, we can safely deduce that Σ_1 is not a tautology. Let us define $\mathcal{A}_2$, another complete assignation function for Σ_1 as $\mathcal{A}_2(a) = \top, \mathcal{A}_2(b) = \top, \mathcal{A}_2(c) = \top$. $\mathcal{A}_2$ is clearly a model of Σ_1. As a model has been found, it is possible to deduce that Σ_1 is consistent.

Let us define Σ_2 as the formula $a \wedge \neg a$ and every possible assignation functions for Σ_2: $\mathcal{A}_3(a) = \top$, $\mathcal{A}_4(a) = \bot$. As none of them provide a model for Σ_2, Σ_2 is said to be unsatisfiable.

Let us define Σ_3 as the formula $a \vee \neg a$ and every possible assignation functions for Σ_3: $\mathcal{A}_3(a) = \top$, $\mathcal{A}_4(a) = \bot$. Each of the possible assignation functions is a model for Σ_3, therefore Σ_3 is a tautology.

End of Example 1.2

Once a variable is assigned, multiple simplifications can be operated. Let us define the notation $\Sigma|_a$ to represent the formula Σ where the variable a has been assigned to $\top$ and $\Sigma|_{\neg a}$ to represent the formula Σ where the variable a has been assigned to $\bot$.

Example 1.3: compute $\Sigma|_a$ Let us define Σ as $C_1 \wedge C_2 \wedge C_3 \wedge C_4$ where $C_1 = a \vee b \vee c$, $C_2 = \neg a \vee b$, $C_3 = \neg b \vee c \vee d$, $C_4 = d \vee e \vee f$. If we assign a to $\top$, we have to compute $\Sigma|_a$ from Σ. It is possible to remove C_1 from Σ as a appears in it. C_2 becomes b as $\neg a$ is evaluated to $\bot$ and therefore assign $\top$ to b. This implies that we have to compute $\Sigma|_a|_b$. The final result is $(c \vee d) \wedge (d \vee e \vee f)$.

End of Example 1.3

It is possible to define some functions on assignations. Let us first give an absolute order on the elements of $\Pi(\Sigma)$. Next, we define the function π defined as $\pi : \Pi(\Sigma) \rightarrow [0..n]$, the function that maps each of the n variables of Σ to an integer. Given a complete assignation $\mathcal{A}$, we can create the Boolean vector $v_{\mathcal{A}}$ as the vector containing at position i, the truth value assigned to variable a such that $\pi(a) = i$. Now that assignation can be interpreted as a vector, we can use some common operations on them, such as the hamming distance. When we have two vectors of Boolean variables $\overline{x} = \langle x_1, x_2, x_3 \rangle$ and $\overline{y} = \langle y_1, y_2, y_3 \rangle$, we can define the Hamming distance function

$$\begin{aligned} \mathcal{H} : \{\top, \bot\}^n, \{\top, \bot\}^n &\rightarrow \{x | x \in \mathbb{N}, x \leq n\} && (1.1) \\ \mathcal{H}(\overline{x}, \overline{y}) &= \text{cardinality}(\{x_i \neq y_i, 0 \leq i \leq n-1\}) && (1.2) \end{aligned}$$

Example 1.4: Hamming distance Let $\Sigma = (a \wedge (\neg b \Rightarrow c))$. We have that $\Pi(\Sigma) = \{a, b, c\}$ and let us define $\pi(a) = 0, \pi(b) = 1, \pi(c) = 2$. Let $\mathcal{A}_1(a) = \top, \mathcal{A}_1(b) = \top, \mathcal{A}_1(c) = \bot$ and $\mathcal{A}_2(a) = \bot, \mathcal{A}_2(b) = \top, \mathcal{A}_2(c) = \top$.

The Hamming distance between the assignation $\mathcal{A}_1$ and $\mathcal{A}_2$ is 2 as $\mathcal{A}_1(a) \neq \mathcal{A}_2(a)$, $\mathcal{A}_1(b) = \mathcal{A}_2(b)$ and $\mathcal{A}_1(c) \neq \mathcal{A}_2(c)$.

End of Example 1.4

1.1.2 Propositionnal calculus

Propositional calculus, as defined by [Encyclopædia Britannica, 2013], is a symbolic system of treating compound and complex propositions and their logical relationships. In the case of Boolean logic, propositions are Boolean formulae and their relationships are defined by the different rules, such as the resolution rule.

Normal forms

In order to have a set of rules on Boolean formulae, it is easier to define a form that every formula must have. The most well known normal forms are the *conjunctive normal form* (or CNF for short) and the *disjunctive normal form* (or DNF for short).

Both CNF and DNF uses the concept of literal. A literal is a variable (a) or its negation ($\neg a$). A literal can be negated: the negation of a is $\neg a$ and the negation of $\neg a$ is $\neg\neg a = a$. For ease of the explanation, we consider that both assignation functions $\mathcal{A}_1(a) = \top$ and $\mathcal{A}_2(a) = \bot$ on the variable a also defines assignation functions on the literals a and $\neg a$: $\mathcal{A}_1(a) = \top, \mathcal{A}_1(\neg a) = \bot$ and $\mathcal{A}_2(a) = \bot, \mathcal{A}_2(\neg a) = \top$.

For the DNF, the main object is a cube: a disjunction of literals $(a \wedge \neg b)$. In this manuscript, cubes will be written as K with some subscripts to make the distinction between the different cubes. A cube is said unsatisfied when at least one literal is evaluated to $\bot$. Therefore, if a cube is empty it can only be satisfiable. DNF formulæ are composed of conjunctions of cubes: $K_1 \vee \ldots \vee K_n$. The DNF formula Σ is unsatisfied if there exists at least one assignation such that every cube in Σ is unsatisfied.

The main object in a CNF is a clause: a conjunction of literals $(a \vee \neg b)$. In this manuscript, clauses will be written as C with some subscripts to make the distinction between the different clauses. A clause is said satisfied if there exists at least one assignation such that at least one of the literal can be evaluated to $\top$. Therefore, if the clause is empty it can only be unsatisfiable. CNF formulæ are composed of disjunction of clauses: $C_1 \wedge \ldots \wedge C_n$. The CNF formula Σ is satisfied if there exists at least one assignation such that every clause in Σ is satisfied. Therefore, if Σ does not have any clause, Σ is a tautology.

We can consider that a clause is a set of literals. Therefore, we can also use the operator of sets on clauses like the cardinality operator $\#$. When $\#C = 1$, C is said to be a unit clause . If $\#C = 2$, C is said to be a binary clause. In a CNF, a formula is a conjunction of clauses $\Sigma = C_1 \wedge \ldots \wedge C_n$ and can also be considered as a set of clauses.

Now that every notion needed to define the main subject of this thesis, let us define the SAT problem formally. The problem consisting of determining if a Boolean CNF formula Σ is satisfiable is called the *Boolean satisfaction problem* (SAT).

When in a set of clauses a literal appears with only 1 phase (either l or $\neg l$), it is said to be a pure literal.

A clause C_1 subsumes a clause C_2 if $\Pi(C_1) \subset \Pi(C_2)$ *i.e.*, if every literal of C_1 is also appearing in C_2. When such facts are known, it is quite useful as $C_2 \models C_1$. Therefore, if one searches the satisfiability of C_1 and knows that C_2 is unsatisfiable, it is possible to conclude that C_1 is also unsatisfiable.

There are some other well known normal forms such as the *and-inverter graph* (AIG). They allows representation for circuits and manipulation of those. Another normal form is *decomposable negation normal form* (DNNF) [Darwiche, 2001]. Such normal form provides some interesting properties such as deciding the satisfiability in linear time and counting the number of models.

Conversion of general Boolean formula to CNF

It is possible to convert any general Boolean formula to a CNF formula Σ. A simple (but inefficient) way is to take every possible assignation function $\mathcal{A}_p$. Whenever $\mathcal{A}_p$ is evaluated to $\bot$, it defines a clause $C_p = a_1 \vee \ldots \vee a_n$ such that a_i is positive if $\mathcal{A}_p(a_i) = \top$, and negative otherwise. The CNF formula will be obtained by a conjunction of every clause C_p.

Example 1.5: Formula to CNF Let us define $\Sigma = a \otimes b$. The two assignation that can be evaluated to $\bot$ are $\mathcal{A}_1(a) = \top, \mathcal{A}_1(b) = \top$ and $\mathcal{A}_2(a) = \bot, \mathcal{A}_2(b) = \bot$. From those, we can create $C_1 = a \vee b$ and $C_2 = \neg a \vee \neg b$. Therefore we can conclude that $\Sigma = C_1 \wedge C_2$.

As said previously, such transformation can be quite inefficient. Indeed, let us consider the unsatisfiable formula $\Sigma = (a \vee b) \equiv \neg(a \vee b)$. As the formula is unsatisfiable, it will create as many clauses as possible assignations. However, as $\Sigma \models \bot$, Σ can be simplified to one empty clause.

End of Example 1.5

The transformation of a general boolean formula to a CNF can be done in polynomial time and space (number of propositional variable). An algorithm is provided in [Siegel, 1987].

Resolution

In order to use a CNF to reason about logic, a propositional proof system is needed. A propositional proof system is an effective method to verify a proof of unsatisfiability [Cook and

Reckhow, 1979]. To define a propositional proof system, one or several derivation rules are needed. From the input formula, applying the given derivation rules and only the given derivation rules ensure that the result is equi-satisfiable. A proof is a succession of application of some rules. Once those rules are known, it is possible to automate their use and therefore automate reasoning about logic. Such rule can be the resolution rule, defined in Equation 1.3. This equation depicts the two input clauses (above the line), and the result (under the line).

$$\frac{(a_0 \lor \ldots \lor a_n \lor l) \land (b_0 \lor \ldots \lor b_m \lor \neg l)}{a_0 \lor \ldots \lor a_n \lor b_0 \lor \ldots \lor b_m} \tag{1.3}$$

Let us define $\Sigma_1 \vdash_{\mathcal{R}} \Sigma_2$ the symbol meaning that it is possible to obtain (to derive) Σ_2 from applying the resolution rule on Σ_1. It is possible to prove that the resolution rule is adequate, that is if $\Sigma \vdash_{\mathcal{R}} C$, then $\Sigma \models C$. However, it is impossible to prove that the resolution rule is complete. Indeed, $p \land \neg q \models q$ but we have not $p \land \neg q \vdash_{\mathcal{R}} q$. This could seems problematic but fortunately, the resolution rule is weak-complete, that is if $\Sigma \land \neg C \models \bot$ then $\Sigma \land C \vdash_{\mathcal{R}} \bot$. As the general case $\Sigma \models C$ can be proven through $\Sigma \land \neg C \models \bot$, this weak-completion of the resolution rule is indeed, not very problematic.

We define the resolution operator $\otimes_{\mathcal{R}}$ as a binary operator providing as result the resolution of the operands.

Example 1.6: a proof sample Let Σ be $(a \lor b \lor c) \land (\neg a \lor b) \land (b \lor \neg c)$ and we want to prove that $\Sigma \models b$. The resolution rule being weak-complete, we will try to prove $\Sigma \land \neg b \models \bot$

$$\cfrac{\cfrac{\cfrac{a \lor b \lor c \quad \neg a \lor b}{b \lor c} \quad b \lor \neg c}{b} \quad \neg b}{\bot}$$

End of Example 1.6

It is interesting to see that the result of a resolution can be used to further. Indeed, it is possible that the result of a resolution subsumes one of the operand of the resolution. C_1 is said to self-subsume with C_2 if $C_1 \otimes_{\mathcal{R}} C_2$ subsumes C_1

Example 1.7: Self-subsuming clause Let $\mathcal{C}_1 = a \lor b \lor c \lor d$ and $\mathcal{C}_2 = a \lor b \lor \neg c$. The resolvent between $\mathcal{C}_1$ and $\mathcal{C}_2$ is $\mathcal{C}_1 \otimes_{\mathcal{R}} \mathcal{C}_2 = a \lor b \lor d$.

End of Example 1.7

Thanks to the resolution rule being weak-complete and adequate, it is possible to prove some well known facts such as the DeMorgan laws, depicted in Equation 1.4 and 1.5. Those laws are quite helpful as we can clearly see that proving that a CNF Σ is unsatifiable is equivalent to prove that the DNF $\neg\Sigma$ is a tautology.

$$\neg(p \lor q) \equiv (\neg p \land \neg q) \tag{1.4}$$

$$\neg(p \land q) \equiv (\neg p \lor \neg q) \tag{1.5}$$

1.2 Computer science

We define computer science[1] as the scientific fields related to automated computing. Such fields are algorithmic, complexity theory, language design, networking, security and many others. Let us first formally define automated computing through the Turing machine. From there, we will wander to complexity notions and finally, artificial intelligence.

1.2.1 Turing machine

The following description of Turing machine is adapted from [Wolper, 2006]. Turing machines, defined in [Turing, 1936], is the basic representation of computers. They are made from multiple parts. First, an infinite memory tape that can be written on through the tape head. Each cell of the tape may contain a symbol from a given alphabet. Secondly, a finite set of states and a transition function represents the behaviour of the machine. This set includes the initial state and the accepting states. Finally, it also contains a transition function that for each possible state of the machine and each possible input on the tape, provides the next state of the machine, the symbol that will be put on the tape and the direction –to the right ($\rightarrowtail$) or to the left ($\leftarrowtail$)– the tape head will take.

The execution of Turing machines can be described as following. First, the input word is on the n first cells of the tape and the blank symbol is on every other cells. The tape head reads the symbol, and according to the state of the machine it writes another symbol on the cell and the tape head is moved. The input word is accepted if the machine reaches an accepting state.

A Turing machine is formally defined by:

- the states $\mathcal{S}$ this machine can take
- an alphabet Γ
- α the white symbol $\alpha \in \Gamma$
- the entry symbols $\Upsilon \subset \Gamma\backslash\alpha$
- s_0 the initial state
- $\mathcal{F} \subset \mathcal{S}$ the accepting states
- $\phi : \{\mathcal{S}, \Gamma\} \to \{\mathcal{S}, \Gamma, \{\leftarrowtail, \rightarrowtail\}\}$ a transition function and the set of accepting state $\Omega \subset \mathcal{S}$.

A Turing machine T is said to be universal if every possible Turing machine can be emulated through T.

1.2.2 Complexity

Once a Turing machine $\mathcal{M}$ has been established for a given type of problems, we might want to quantify the time needed to solve those problems, *i.e.* the number of transitions made. Let $\mathcal{M}$ halt for each input $\vec{x}$, n the size of $\vec{x}$ and $t_{\mathcal{M}}(\vec{x})$ the number of operations needed to perform the computation. We can compute the general function $t(n)$ that will provide the number of

[1]Some argue that computer science is not a right name, as if we would call physics "hammer science"

operations needed in the *worst case scenario*. This defines the time complexity of a Turing machine.

To compare two Turing machines, we can use their time complexity but sometimes, only the asymptotic value (where n grows infinitely) is considered. To do this, we can use the $\mathcal{O}$-notation. A function $g(n)$ is said to be $\mathcal{O}(f(n))$ if there exists some constants c and n_0 such that $\forall n > n_0, g(n) \leq cf(n)$. Using this notation provides different operations on it

- $\mathcal{O}(c_1 n^k + \ldots + c_k n + c_{k+1}) = \mathcal{O}(n^k)$
- $\mathcal{O}(t(n) + t(n)) = \mathcal{O}(t(n))$
- $\mathcal{O}(t_1(n) + t_2(n)) = \mathcal{O}(max(t_1(n), t_2(n))$
- $\mathcal{O}(t(n)t(n)) = \mathcal{O}(t(n)^2)$

It is possible to characterize algorithms according through the $\mathcal{O}$-notation.

- Constant time $\mathcal{O}(1)$: access to an element at a given position in the memory.
- Linear time $\mathcal{O}(n)$: sum of two vectors containing n elements.
- Polynomial time $\mathcal{O}(n^x)$: multiplication of two matrices of size n.
- Exponential time $\mathcal{O}(x^n)$: generating every subset of a set containing n elements.

1.2.3 NP Completeness

It is also possible to characterize the problem themselves instead of algorithms.

Decision problems that can be solved in polynomial time by a deterministic Turing machine are members of the P complexity class. Decision problems that can be verified in polynomial time by a deterministic Turing machine are members of the NP complexity class. Decision problems $\mathcal{P}$ such that any problem from NP can be reduced in polynomial time to $\mathcal{P}$ are NP-complete. Problems $\mathcal{P}$ such that any NP-complete problem can be reduced to $\mathcal{P}$ in polynomial time are NP-hard. This definition of NP-hard does not require that the solution can be verified polynomial time by a deterministic Turing machine.

From those definitions, one of the most interesting questions that have not been resolved yet is to know whether $P = NP$ or $P \subset NP$.

The first problem that has been proven NP-complete is the satisfiability of Boolean formula. The proof [Cook, 1971] by Stephen Arthur Cook was done in two steps. First, a polynomial function that verifies an instantiation is provided and secondly, every NP problem is proven to be reducible in polynomial time to SAT. The first part is trivial. As for the second part, it is done by defining a polynomial function that maps every couple $w, \mathcal{L}$ to a SAT instance where w is a input word for the language $\mathcal{L}$. If $\mathcal{L} \subset NP$ accepts w if and if only there is a non-deterministic Turing Machine $\mathcal{M}$ that accepts w. From such machine $\mathcal{M}$ and word w, we create the corresponding SAT instance, satisfiable if and only if $\mathcal{M}$ accept w. The proof proposes the creation of such SAT instance in polynomial time, making the problem of knowing whether a Boolean formula is satisfiable a NP-complete problem. As long as $P = NP$ is unproved, we can assume that every algorithm capable of solving the SAT problem is in exponential time.

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers thinks . . . machines with minds, in the full and literal sense." [Haugeland, 1985] "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." [Bellman, 1978]	"The study of mental faculties through the use of computational models." [Charniak and McDermott, 1985] "The study of the computations that make it possible to perceive, reason, and act." [Winston, 1992]
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." [Kurzweil, 1990] "The study of how to make computers do things at which, at the moment, people are better." [Rich and Knight, 1991]	"Computational Intelligence is the study of the design of intelligent agents." [Poole et al., 1998] "AI . . . is concerned with intelligent behavior in artifacts." [Nilsson, 1998]

Table 1.3: AI definitions categories as provided in [Russell and Norvig, 2003]

1.3 Artificial Intelligence

There are a lot of different topics one could learn to achieve artificial intelligence. In [Russell and Norvig, 2003] the authors provide introduction to different -but not all- areas involved with this field. Their result is a thousand pages book, giving only a glimpse of different techniques such as computer vision, reasoning, planification, natural language processing and many others. Research in artificial intelligence goes way beyond computer science. Indeed, some techniques have been directly influenced by biology or even philosophy.

In their book, [Russell and Norvig, 2003] provide four different categories of definitions for artificial intelligence shown in Table 1.3. We can see that their definitions diverge and that some use the concept of *intelligence* which is also subject to multiple definitions.

As we can see, there are a lot of different definitions of artificial intelligence. Instead of defining what artificial intelligence tries to mimic, we can define artificial intelligence by the kind of problems it tries to solve. A common point between every field labelled as artificial intelligence is their effort to solve problems which are at least NP-complete. Therefore, in this thesis, the author defines artificial intelligence as the scientific field of providing algorithms to solve problems that are at least NP-complete such as the Boolean satisfaction problem.

1.4 Conclusion

In this chapter, we presented briefly some concepts needed for a better understanding of this thesis. The Boolean satisfaction problem (SAT problem) can be solved using algorithms through propositional calculus. The next chapter presents some algorithms to solve it.

CHAPTER 2

SAT solvers

It's a kind of magic

—Roger Taylor

As seen in previous chapter, given an input Boolean CNF formula Σ, knowing whether Σ has a model is an NP-complete problem. However, as solving the Boolean satisfiability problem is helpful in a lot of domains, multiple algorithms have been proposed. Programs that are composed on such algorithms and having as goal providing a solution for the SAT problem are called SAT solvers. The content of this chapter is the following: first, we are going through some methods or algorithms to solve the Boolean satisfiability problem. Second, we take a closer look at the algorithm which has been studied through this thesis. Each part is described in order to provide the best comprehension of this kind of solver, called CDCL. Finally, different implementations are presented.

2.1 Main SAT solving algorithms

In this section, we review some of the most well-known methods that have been proposed through time to solve the Boolean satisfiability problem.

2.1.1 Truth tables

One of the first method proposed to solve the SAT problem is the generation of truth tables. The algorithm to generate the truth tables is extremely simple. It creates every possible instantiation and whenever an instantiation provides a model, the formula is proven *satisfiable*. If no instantiation provides any model, the formula is proven *unsatisfiable*. An example of such table is provided in Figure 2.1 on the formula $\Sigma = (a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$.

Even though this method is really easy to implement, it has some major drawbacks. It completely ignores the problem, its structure and its relative information and a complete instantiation is needed to evaluate the truth value of the function. Moreover, the asymptotic complexity of this problem, $\mathcal{O}(2^n)$ where n is the number of variables, is reached as soon as the formula is *unsatisfiable*.

a	b	c	Σ	a	b	c	Σ
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$

Table 2.1: Truth table for the formula $\Sigma = (a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c)$

Example 2.1: Simple UNSAT formula Let us suppose that we want to know

a	b	c	Σ	a	b	c	Σ
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$

if the formula $\Sigma = (a \vee c) \wedge (\neg a \vee c) \wedge (\neg c \vee \neg b) \wedge (b \vee \neg c)$. We can easily see that this formula is unsatifiable as the first two clauses implies that $c = \top$ (indeed, $(a \vee c) \otimes_{\mathcal{R}} (\neg a \vee c) = c$). But the two last clauses implies that $c = \bot$ (indeed, $(\neg c \vee \neg b) \otimes_{\mathcal{R}} (b \vee \neg c) = \neg c$) making the formula unsatisfiable. However, the generation of the whole table is needed to conclude the unsatifsiability of Σ through the truth-tables method.

End of Example 2.1

2.1.2 Davis, Logemann, Loveland's algorithm

One of the first algorithm proposed was proposed by Martin Davis and Hilary Putnam [Davis and Putnam, 1960]. Their main idea is to use the $\otimes_{\mathcal{R}}$ operator in order to remove variables until no variable can be removed or the empty clause is generated. This method has the drawback of needing a lot of memory to store all generated clauses.

The Davis, Logemann and Loveland's procedure (or DLL for short) [Davis et al., 1962] tried to avoid the generation of the clauses of DP. DLL is a recursive algorithm on the formula, depicted in Algorithm 2.1. The main idea is as follows: first, we try to simplify the formula Σ by assigning literals appearing in unit clauses ($\{l_u | C_u = l_u, C_u \in \Sigma\}$) and pure literals ($\{l_p | l_p \in \Sigma, \neg l_p \notin \Sigma\}$). Then, from the simplified formula, different checks are made to see if the answer has been found or if a recursive call must be made. It is guarantee to end as the number of non-assigned literals decreases with the recursive calls through the conjunction of the formula and a unit clause.

It is possible to graphically depict the search made by the algorithm through a tree structure as in Figure 2.1. Each node of the tree represents the current state of the formula given at the root, simplified with the assignation given on the path between the node and the root. This tree is constructed by depth- and left-first.

Algorithm 2.1: DLL algorithm

```
Input  : $\Sigma$ a Boolean formula in conjunctive normal form
Output: $\top$ if $\Sigma$ is satisfiable; $\bot$ if $\Sigma$ is unsatisfiable
begin
    foreach $l \in \{l_u | C_u = l_u, C_u \in \Sigma\} \cup \{l_p | l_p \in \Sigma, \neg l_p \notin \Sigma\}$ do
        $\Sigma \leftarrow \Sigma|_l$ ;
    done
    if $\Sigma$ contains an empty clause then
        return $\bot$;
    elif $\Sigma = \emptyset$ then
        return $\top$;
    fi
    choose a literal $l$ from $\Sigma$ ;
    return DLL($\Sigma \wedge l$) $\vee$ DLL($\Sigma \wedge \neg l$)
end
```

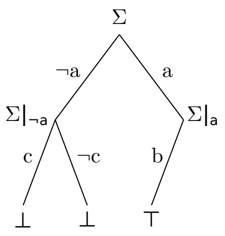


Figure 2.1: DLL algorithm depiction for the formula $\Sigma = (a \vee c \vee b) \wedge (a \vee c \vee \neg b) \wedge (a \vee \neg c \vee b) \wedge (a \vee \neg c \vee \neg b) \wedge (\neg a \vee b)$.

Example 2.2: DLL in action Let us consider following instance; solved with Algorithm 2.1:

$$a \vee b \vee c, \quad a \vee \neg b \vee \neg c, \quad \neg d \vee e, \quad \neg g,$$
$$\neg d \vee f \vee \neg e, \quad e \vee \neg f, \quad e \vee a, \quad \neg e \vee f \vee g$$

At first, there are two pure literals: a and $\neg d$ and a unit clause: $\neg g$. We set them to true: $a = \top, b = \bot, g = \bot$. We can remove the clauses containing the pure literals as they are already satisfied, remove those containing $\neg g$ and remove g from clauses that contain it. Once Σ is fully simplified, the left clauses are: $e \vee \neg f$ and $\neg e \vee f$. As there are still clauses left, we must choose one of the variables left, e for instance, and make a recursive call. The call is made on the formula $(e \vee \neg f) \wedge (\neg e \vee f) \wedge e$. After simplification, $\Sigma = \emptyset$ and therefore $\top$ is returned. If we wanted to represent this search by a tree, it would be only a branch annotated by e since it is the only chosen literal.

End of Example 2.2

Example 2.3: DLL on an unsatisfiable instance Let us consider following instance:

$$a \vee b \quad \neg a \vee \neg b$$
$$\neg a \vee b \quad a \vee \neg b$$

No simplification can be done, therefore a recursive call is made with a as literal. After the simplification $\Sigma|_a$, we obtain $b \wedge \neg b$. As b is a unit clause it is simplified, leading to an empty clause and therefore the value $\bot$ is returned. As the call with the formula $\Sigma \wedge a$ failed, we must try $\Sigma \wedge \neg a$. However, this leads again to the formula $b \wedge \neg b$ and returns $\bot$. As both branches have been tested and failed, the main call returns $\bot$, stating that this formula is *unsatisfiable*.

End of Example 2.3

As we can see, choices are needed in order to solve the formula, those can be made using heuristics. Those heuristics can make a huge difference in term of computation needed to obtain a solution as shown in the next Example.

Example 2.4: Branching simplification Let us suppose that we want to solve the formula $\Sigma = (a \vee \Sigma') \wedge (\neg a \vee \Sigma')$, $a \notin var(\Sigma')$. We can easily see that a has no impact on the satifiability of Σ. However, let us now suppose that $\Sigma = (a \vee \Sigma') \wedge (\neg a \vee \Sigma'')$, $var(\Sigma') \cap var(\Sigma'') = \emptyset, a \notin var(\Sigma'), a \notin var(\Sigma'')$, branching on a simplifies the formula.

End of Example 2.4

Different heuristics are possible to select the variable. Static heuristics are one of those. They create an order on variables that never changes during the search procedure. It can be based on the occurrence of a given literal on the initial formula, or the size of the clauses in which the literal appears or any other information. They can be useful if the person coding the order has a good knowledge of the formula that needs to be solved. However, static heuristics are not widely used anymore as per definition, they can not evolve during the search and therefore can not use any information that could have been discovered during the search.

Another class of heuristics is based on *look-ahead* and is very useful on random formulæ.. This class of heuristic computes the impact of choosing the literal x by computing $\Sigma|_x$. Through this computation, some literals $l_0, \ldots, l_i$ are propagated. The impact of x needs to be combined with the impact of $\neg x$ in order to create a well-balanced tree. As both $\Sigma|_x$ and $\Sigma|_{\neg x}$ are computed, this heuristic provides as side effect the creation of unit clauses. Indeed, if l_α is propagated by both x ($\Sigma \wedge x \models l_\alpha$) and $\neg x$ ($\Sigma \wedge \neg x \models l_\alpha$), it is possible to conclude that $\Sigma \models l_\alpha$ [Le Berre, 2001]. One instantiation of look-ahead heuristic is proposed by the author of [Freeman, 1995]. The number of reduced free variables is used as impact factor $\mathcal{I}_x$. The combination of the impact factors $\mathcal{I}_x$ and $\mathcal{I}_{\neg x}$ is done by the formula $1024 \times \mathcal{I}_x \times \mathcal{I}_{\neg x} + \mathcal{I}_x + \mathcal{I}_{\neg x}$.

Example 2.5: Look-ahead Let us define Σ by:

$$\begin{array}{cc} a \vee c & \neg a \vee c \\ \neg a \vee \neg c \vee b & \neg d \vee e \vee g \end{array}$$

According to the formulae provided in [Freeman, 1995], $\mathcal{I}_a$ is 2 as $\Sigma \wedge a \models c, b$, $\mathcal{I}_{\neg a}$ is 1 as $\Sigma \wedge \neg a \models c$, $\mathcal{I}_d$ is 0, $\mathcal{I}_{\neg d}$ is 0. The combination of the impact factor gives 2051 for the variable a and 0 for d. Using such heuristic, between a and d, the algorithm choose a as branching variable.

End of Example 2.5

A second instantiation of the look-ahead heuristic is proposed by the author of [Li, 1999]. Given a formula Σ, $\Sigma|_x$ will simplify some clauses to binary clauses (for instance $(a \vee b \vee \neg x)|_x = a \vee b$). The set of such simplified clauses is $\mathcal{S}_x$. We also define $\mathcal{N}_a^n$ as the number of clauses of the form $x_1 \vee \ldots \vee x_n \vee a$ in Σ. The score of a variable is defined by $s(x)$ inspired by [Freeman, 1995], shown in the following equation where C is some constant.

$$s(x) = 1024 \times w(x) \times w(\neg x) + w(x) + w(\neg x) + 1$$

$$w(x) = \sum_{(u \vee v) \in \mathcal{S}_x} f(\neg u) + f(\neg v)$$

$$f(x) = \begin{cases} \mathcal{N}_x^2 & \text{if } \mathcal{N}_x^2 > C \\ \sum_i 5^{2-r} \mathcal{N}_x^i & \text{otherwise} \end{cases}$$

A third instantiation of the look-ahead heuristic is proposed by the authors of [Dubois and Dequen, 2001]. The score obtained for the variable x is obtained from the following formula:

$$h(x, i) = \begin{cases} \sum_{(u \vee v) \in \mathcal{S}_x} h(\neg u, i+1) h(\neg v, i+1) & \text{if } i < M \\ \sum_{(u \vee v) \in \mathcal{S}_x} (2\mathcal{N}_{\neg u}^1 + \mathcal{N}_{\neg u}^2)(2\mathcal{N}_{\neg v}^1 + \mathcal{N}_{\neg v}^2) & \text{otherwise} \end{cases}$$

2.1.3 Local search

Next to complete algorithms exist a great number of incomplete algorithms. Many of those use the local search schema. The main difference with complete algorithm is that almost every local search algorithm is not able to prove the unsatisfiability of a formula. During the computation, the algorithm tries to generate a model for the formula through an instantiation vector.

A schema for local search is provided in [Hentenryck and Michel, 2009] and depicted in Algorithm 2.2. First, an instantiation vector $\overrightarrow{v}$ is generated. Then, as long as $\overrightarrow{v}$ does not provide a model, it is refined by the following instructions. A set $\mathcal{N}$ of candidates are generated by applying a modification on $\overrightarrow{v}$. This set is called the neighbours of $\overrightarrow{v}$. From $\mathcal{N}$, the algorithm removes some values to obtain the set $\mathcal{V}$ of "valid" neighbours. Here, valid designates assignations vectors that do not break some predefined rules. Finally, from $\mathcal{V}$, a new candidate is selected. As the completion of the algorithm is not guarantee, a limit m on the number of tries is usually enforced.

Algorithm 2.2: general local search schema

```
   Input  : $\Sigma$ a Boolean formula in conjunctive normal form
   Input  : $m$ the maximum number of iteration allowed
   Output: $\overrightarrow{v}$ a vector of instantiation
 1 begin
 2     Generate initial instantiation vector $\overrightarrow{v}$ ;
 3     $i \leftarrow 0$ ;
 4     while $i < m \wedge \Sigma$ is not satisfied by $\overrightarrow{v}$ do
 5         Generate set of instantiation vector $\mathcal{N}$ from $\overrightarrow{v}$ ;
 6         Create $\mathcal{V}$ by filtering vectors from $\mathcal{N}$ ;
 7         $\overrightarrow{v} \leftarrow$ best candidate from $\mathcal{V}$ ;
 8         $i \leftarrow 1 + i$ ;
 9     done
10 end
```

Heuristics have been defined for each of the important step of local search algorithm (initial generation, neighbours generation, filter and candidate selection). Some of those are well known even outside the local search community. For instance, the taboo search [Glover, 1989] can be used as filter heuristic. In order to be valid, a potential candidate is forbidden to have been used as one of the n previously selected value. The neighbours generation can be done by swapping the value of some selected variables. Those variables can be selected according to the 'most violating' principle: for each clause C unsatisfied, increase a counter of every variable used in C. The variable used as candidate are those with the highest value.

More information about local search algorithms can be found in [Kautz et al., 2009].

2.1.4 Conflict driven clause learning

The general idea behind conflict driven clause learning (CDCL for short) is to increase the power of the DLL procedure by the explicit use of the $\otimes_{\mathcal{R}}$ operator. Whenever a conflict rises, it means that there are two clauses $C \vee x$ and $C' \vee \neg x$ that are both unsatisfied. We can apply the resolution operator on those clauses to obtain a new clause $C \vee C'$ (we call this clause the reason) that helps us to avoid the conflict in the future. This conflict is avoided by reverting decisions that were made and lead to the conflict. We call this reversion of decision *backjumping* or *non-chronological backtracking*.

Moreover, in order to improve the practical efficiency, restarts are made [Gomes et al., 2000]. Finally, the number of generated clause rises and slows down the algorithm. Therefore, those can be deleted according to a given periodicity.

The DLL procedure was improved by the help of non-chronological backtracking in `relsat` [Bayardo and Schrag, 1997]. The `grasp` solver [Marques-Silva and Sakallah, 1997] introduced the concept of learnt clauses. Based on those work, the solver `chaff` introduced the watched literals. Finally, the `MiniSat` solver used those technique to obtain the well known and competitive solver widely used.

The main changes comparing to the DLL procedure is the generation of the learnt clauses, their deletion, backjumping and restarts. As the backjumping are performed because of the reasons and the reason is a reformulation of elements present in the initial formula, introducing them does not change the completeness of the algorithm. However, the addition of restarts and deletion of reasons might lead to an incomplete algorithm, depending on the periodicity of those actions. If the periodicity of both of those actions is random or constant, the resulting algorithm is not complete as it might return to its initial state. But if the periodicity of one of those action is increasing, we keep the completeness of the algorithm as explained in the following paragraphs.

Let us suppose that the algorithm has its n^{th} restart after x_n events, an event being a conflict, a second or assignation. If the sequence of $x_i, i \in \mathbb{N}$ is crescent, there is a j such that x_j is greater than the number of events needed to solve the formula.

Now, let us suppose that the algorithm has its n^{th} reasons deletion after x_n events and r_n being the number of reasons left before the deletion of some reasons. If the sequence $r_i, i \in \mathbb{N}$ is crescent, there is a value j such that r_j is greater than the number of possible reasons. If every reason is generated, we find either a solution or an empty reason.

2.2 CDCL in-depth

The CDCL algorithm is explained in detail in this section. As the complete algorithm is too big to be explained at once, each main phase has its own description. As some data structures are used through the different parts, a glossary explaining each of such data structures is present at page 44. However, in order to ease the reading, each time a new variable is used, a short description is made.

The problem that we are trying to solve is to find either a complete instantiation such that the formula is evaluated to $\top$, or prove that no such instantiation exists. Algorithm 2.3 is the result from years of research. Let us explain it step by step. The main idea is the following: as long as no conflict appears, we assign a truth value to a literal (line 11). From this assignation, values for other variables can be deduced through the unit propagation (line 4). Let us recall that DLL was already using unit propagation (Alg. 2.1, line 3, page 13). Whenever a conflict is detected – every literal of a clause has been assigned and the clause is unsatisfied –, an analysis of the conflict is performed (line 6). This analysis provides us a new clause representing the cause of the conflict that is added to the set of new clauses (line 9). This new clause can be referred to as *nogood* or *learnt clause*. However, the analysis algorithm may generate a clause that can be simplified. Therefore, an attempt is made to simplify the clause. (line 7). Once the simplification process is made, the algorithm reverts some of its state and modifies it in order to avoid this last conflict and continues the search (line 8). A clause management mechanism can be applied as the size of the set of new clauses has an effect on the speed of the algorithm (line 13). Finally, the evaluation of the restart condition is made (line 14). If the condition is evaluated to $\top$, the algorithm performs a backjump to the root of the search tree.

There are two possible conditions leading to the termination of the algorithm. The first is whenever every variable has been assigned and no clause is left unsatisfied, giving a solution to the given problem. The second is when the clause generated from the conflict analysis is the empty clause, proving that the given problem is unsatisfiable.

Algorithm 2.3: CDCL scketch algorithm

```
Input  : A CNF formula $\Sigma$;
Output: $\top$ if $\Sigma$ is satisfiable; $\bot$ if $\Sigma$ is unsatisfiable
Data: unsatCls: an unsatified clause
Data: confCls: a clause representing a conflict
Data: end: a Boolean, $\top$ if a solution has been found, $\bot$ otherwise
1  begin
2      end ← $\bot$;
3      while ¬end do
4          Propagations (Alg. 2.7, page 28 );
5          if unsatCls $\neq$ nil then
6              Create confCls (Alg. 2.8, page 32);
7              Simplify confCls (Alg. 2.9, page 36);
8              Backtracking (Alg. 2.10, page 37);
9              Process confCls (Alg. 2.11, page 39) ;
10         else
11             Branching (Alg. 2.6, page 25);
12         fi
13         Clause database management;
14         Restart condition evaluation;
15     done
16 end
```

Now that the general outline of the algorithm has been explained, the details are shown in the following sections.

2.2.1 Branching

Let us begin our in-depth explanation with the branching algorithm and heuristics. This is done in two steps. First, we have to select the variable that will be assigned, then we have to select the value that will be assigned. We have to point out that this element of the solver is more important for the satisfiable formulas. Indeed, when a formula is satisfiable, it usually has more than one model where the value of some of the variables are not important. If the solver branches on one of those variables, the problem might not be simplified and therefore, one could argue that this time was wasted (as shown in Example 2.4).

Once a branch is created, a variable is selected and from this selection, different variables can be assigned. For each of those variable, we define their *level*. The level of the search process is the current number of active decision. A decision is said to be active if it has not been reverted through backtrack. The level of a decision variable is the level of the search process $+1$. The level of a propagated variable is the level of the last decision variable. In our algorithms, the array `varLevel` stores the level for each variable.

Example 2.6: Variable level Let us suppose that we want to solve the formula $\Sigma = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg b \vee \neg a)$. The initial level of the search process is 0 as no choices were made. Let us now suppose that the first decision made is $b = \top$. The level of b is 1. From this assignment, the value of b can not be deduce as a conflict appears: $a \vee \neg b$ and $\neg b \vee \neg a$. Therefore, the search process needs to cancel the decision $b = \top$. If the next decision is $b = \bot$, the level of b is set once more to 1. From this assignation, the search process can deduce the value of a to $\top$. a has now the same level of b, the last decision variable: 1.

End of Example 2.6

Let us now review some heuristics for the variable and value selection.

Variable selection

Different heuristics have been proposed throughout the years. Those can be divided in two categories: look-ahead based (see 14) and activity based.

Activity based heuristics may use the information from the initial formula, but can also use the information discovered during the search. Using this, the solver tries to select the most important variable at the current state of the search.

One of the most well known heuristic for CDCL solver is the Variable State Independent, Decaying Sum (VSIDS) proposed first in [Moskewicz et al., 2001] and revisited many times since. This idea is based on the fact that variables appearing often in the last conflicts are part of a 'difficult' search space. By intensifying the search on such space instead of wandering around, the algorithm can have better performance.

Initially, each variable has a given score. Whenever a conflict is found during the search, we evaluate which variables are part of the reason of this conflict. For each of those variables, their related score is incremented as depicted in Algorithm 2.4. In order to favour the most recent variables, the incrementation value increases after each conflict. In order to avoid problem with the floating number representation, those are contained under e^{100}. Then, when a variable selection has to be performed, the algorithm retrieves the variable with the highest value (Algorithm 2.6, line 11). To have an easy access to the variable with the highest value, a heap `vsidsHeap` is used.

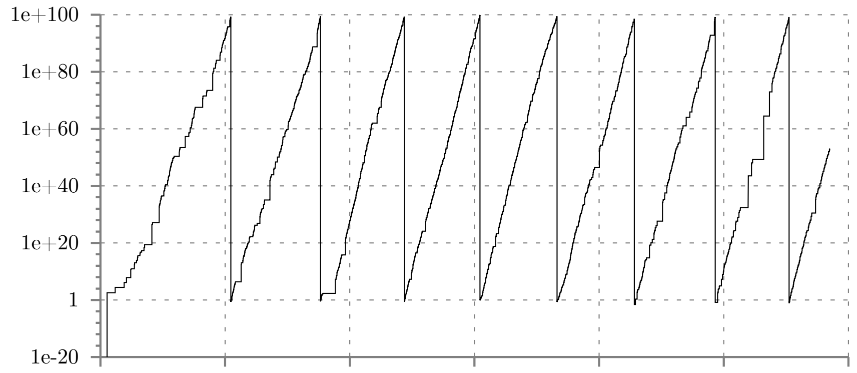


Figure 2.2: Evolution of the VSIDS value for a variable during 38298 conflicts for the instance `manol-pipe-c9`

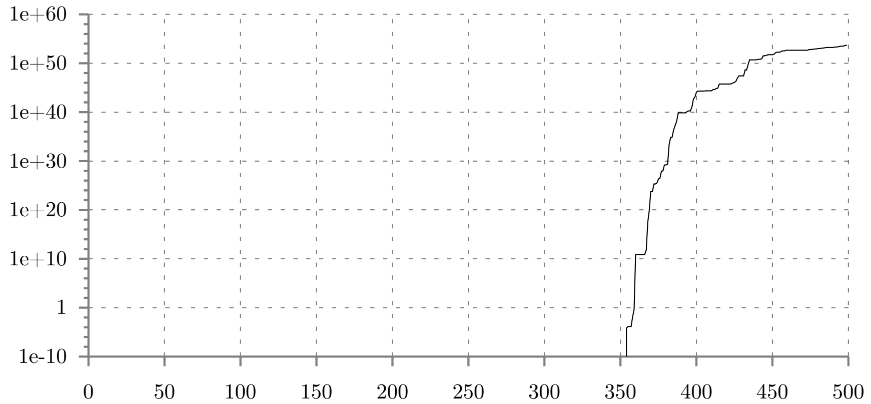


Figure 2.3: Sorted VSIDS value for the first 500 hundreds variables after 38298 conflicts for the instance `manol-pipe-c9`

Figure 2.2 shows the evolution of the VSIDS value of a variable after the 38298 conflicts needed by an implementation to solve the instance `manol-pipe-c9`.[1] We can clearly see that regularly, the value is lowered to keep the values in a manageable range. The stall that can be observed are due to the fact that the variable did not appeared in the conflict resolution. Figure 2.3 shows the values for 500 variables, sorted according to the VSIDS value. We can clearly see that out of those 500 variables, only a few have a high score, allowing to discriminate them easily. Those results are not bounded to the given instance but can be seen in most structured instances.

A second variable selection heuristic is proposed by the authors of [Goldberg and Novikov, 2007]. They use one counter for each variable. Whenever a conflict rises, the value of the counter is incremented by 1 for each literal appearing in each clause responsible for the conflict. Periodically, values are divided by 4. The idea behind this is the same as for the VSIDS: having a high score for variables that have recently appeared in conflicts. The division allows the "aging" process. However, they do not select the unassigned variable with the highest counter. From the set of generated clause, they take the most recent which is not satisfied. From this clause C, the unassigned variable with the highest counter is selected. If there is no generated clause or if every generated clause is satisfied, the chosen variable is simply the one with the highest counter.

A third variable selection algorithm is to randomly select a variable according to a given frequency (Algorithm 2.6, line 3).[2] When no random choice is made, another strategy -deterministic- is needed. This allows the solver to "wander" with this entropy, hoping that this is beneficial for the search.

Algorithm 2.4: CDCL:vsids:update

```
Input  : var: the variable whose vsids activity must be updated
Input  : activity: the array containing the VSIDS score for each variable
Input  : vsidsHeap: a heap containing the variables, sorted according to their VSIDS
         score
Input  : inc: the increment value for the update
begin
    activity[var] ← activity[var] + inc;
    if var is in vsidsHeap then
        update position of var in vsidsHeap;
    fi
    if activity[var] > $e^{100}$ then
        foreach var of the instance do
            activity[var] ← activity[var] $\times e^{-100}$;
        done
    fi
end
```

[1]Information about this instance can be found at `http://www.cril.univ-artois.fr/SAT09/results/bench.php?idev=29&idbench=71771`

[2]It is interesting to note that the latest `MiniSat` implementation has a default frequency of 0. Therefore never making random choices

Value selection

Once the variable is selected, different strategies are available for the value selection. A common heuristic used is the following. We define `polarity` as an array of Boolean representing for each variable the next value that should be assigned. Whenever a backtrack is performed, the value that was assigned to each variable we are reverting is stored in `polarity` (See Algorithm 2.10, line 4). That way, when we have to choose a new value for the variable, the last value used is chosen [Pipatsrisawat and Darwiche, 2007]. The reason behind this, is that we stay as close as possible from the last partial instantiation after a restart. Otherwise, the search space could vary a lot, making the learnt clause less relevant. This heuristic is particularly good with satisfiable formulae, less with unsatisfiable ones.

As for the initialization, there are different possibilities: assign every value to $\top$, $\bot$, or even randomly. A reasonable choice is to assign to $\bot$. The reason is that many Boolean representations of constraints lead to creation of Boolean variable where only one of them must be true. (See next example)

Example 2.7: Constraint modelling Imagine that we are trying to solve the map colouring problem. This problem consists in assigning one of four colours to a country on a map, and no two country sharing a border are allowed to have the same colour. For each of the country, four values are possible, but only one can be selected. A common modelling for such constraint where only one value out of n can be selected is the following. Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ the set of possible values. n Boolean variables are created, where x_i is true if and only if the value v_i has been selected. The clauses needed to represent this constraint are the following:

$$\begin{aligned} \mathcal{C} &= x_1 \vee \ldots \vee x_n \\ \forall_{j=1}^{n} \forall_{k=j+1}^{n} \quad \mathcal{C}_{(j,k)} &= \neg x_j \vee \neg x_k \end{aligned}$$

$\mathcal{C}$ states that at least one value must be selected. The clauses $\mathcal{C}_{(j,k)}$ are needed to limit to only one variable assigned to $\top$. Therefore, if only one of the variables can be assigned to $\top$, it means that every other variable has to be assigned to $\bot$.

End of Example 2.7

Again, in [Goldberg and Novikov, 2007], the authors proposed another technique. Once their literal is chosen (see page 21), they assign it to $\bot$. This comes from the *fail first* principle [Haralick and Elliott, 1980]. If no learnt clause was found, their heuristics uses a function $n(x)$ where x is a literal. This function is defined as the sum of number of binary clauses using the literal x, and for each literal y appearing with x in a binary clause, the number of binary clauses using y. The value selected is the one with the highest value between $n(x)$ and $n(\neg x)$.

During the search, the algorithm needs to know the variable that has been assigned and the order in which they have been assigned. Three kind of operations are needed on this data structure. First, variables must be added to this structure quite often. Therefore, this operation should be executed in constant time. The second operation is to look up the content to know the n^{th} assigned variable. This operation should also be performed in constant time. And finally, the last operation needed is the removal of variables. As the algorithm removes first the last assigned variables, the data structure used is a stack implemented using an array `trail` as depicted in Figure 2.4. No dynamic resizeable structure is needed as the number of variables is known at the start of the search.

This `trail` is filled during the assignation algorithm (Algorithm 2.5) with literals assigned to $\bot$. Variables corresponding to the literals stored in indexes lower than `head` are the one that are already propagated. Those that are stored at position lower than `top` are the ones that have been assigned but not yet propagated.

	0	head	top	nbVar
trail	literals propagated	literals to propagate		

Figure 2.4: Invariant for the `trail`

Algorithm 2.5: CDCL:assignation

```
Input : lit: the literal that must be true
Input : c: the clause that needs lit to be true
Input : varLevel: the level of each variable if the variable is assigned
Input : reason: the array containing for each variable v the clause that forced the
        value of v
Data: v: a variable
begin
    trail[top] $\leftarrow$ $\neg$lit;
    top $\leftarrow$ top+1;
    v $\leftarrow$ variable of lit;
    assign[v] $\leftarrow$ sign of lit;
    if c $\neq$ nil then
        reason[v] $\leftarrow$ c;
    fi
    varLevel[v] $\leftarrow$ level;
end
```

Later, in the conflict analysis (Algorithm 2.8), the reason of an assignation –the clause that needed such assignation– is needed. Those reasons are stored in the array `reason`. If an arbitrary choice is made, that is if it was made at branching, there is no such reason. To point out that there is no reason, the value *nil* is set to the corresponding cell of `reason` (Algorithm 2.6, line 17).

In order to depict the content of `trail`, `varLevel` and `reason`, the following schema is used: $\langle x_0@_{l_0}C_0,\ x_1@_{l_1}C_1,\ \ldots,\ x_n@_{l_n}C_n\rangle$. x_i represents the i-th literal assigned to $\bot$ (`trail[i]`), l_i its level (`varLevel`[variable of `trail[i]`]) and C_i its reason (`reason`[variable of `trail[i]`]). A graphical depiction, called implication graph, can also be proposed. The directed graph has as vertex the literals assigned to $\top$. Edges can have multiple sources but only one destination and represent a clause C that was used to propagate. The destination is the literal propagated and the sources are the literals evaluated to $\bot$ in C. The level of the variable is represented by dotted lines that partition the space, with an indication of the level at the bottom.

Example 2.8: Solver state depiction Let the instance be:

$$\begin{aligned} C_1 = &\quad \neg a \vee b & C_2 = &\quad \neg b \vee d, \\ C_3 = &\quad \neg c \vee e \vee \neg b \vee \neg d \end{aligned}$$

Let us suppose that $a = \top$ was the first decision. This implies through C_1 that $b = \top$ which in turns implies through C_2 that $d = \top$. The second decision is $c = \top$. This decision and the previous implication implies through C_3 that $e = \top$.

As $a = \top$ is the first decision, the level of a is 1. So does also every literal implied before the next decision. Those are b and d. $c = \top$ being the second decision, c has 2 as level. e is here the only literal implied, through C_3, therefore it also has 2 as level.

Now that the information (level, reason and assignations) are known, we can depict the `trail`. Let us recall that in the trail appear the literals assigned to $\bot$. In our example, the trail is $\langle \neg a @_1 nil,\ \neg b @_1 C_1,\ \neg d @_1 C_2,\ \neg c @_2 nil,\ \neg e @_2 C_3 \rangle$.

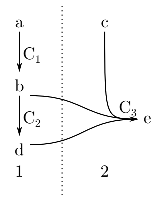


Figure 2.5: Implication graph of the instance presented in Example 2.8

The implication graph is shown in Figure 2.5. We can see the partition of space with the dotted line, representing the different levels. As C_3 is of size 4, there are three sources that leads to $e = \top$. Decision literals present at the top of the graph (a and c) have no incoming edges.

End of Example 2.8

The complete algorithm used for the variable and value selection is depicted in Algorithm 2.6.

2.2.2 Propagations

The propagation algorithm has the following purpose: from a set of variables that have been assigned, find every value that needs to be forced in order for clauses to be satisfied. And if a clause becomes unsatisfied, return it in order to be able to analyse the reason of this conflict. This unsatisfied clause is called the conflict (or conflicting) clause.

This algorithm needs to be fast as a large amount of the computation happens in this code. Therefore, careful coding has been made in the different solvers in order to make the program as fast as possible.

For each newly assigned variable, the algorithm has to check that no clause containing this literal becomes unsatisfied. It checks also if it can deduce the value of some literals.

Algorithm 2.6: CDCL:branching

```
Input  : freq: the frequency on which we want a random variable choice
Input  : polarity: the array containing the next truth value that needs to be selected
Input  : level: the current level of the search
Output: var: the selected variable, if any
begin
    var $\leftarrow$ undefined ;
    if random() $<$ freq $\wedge$ vsidsHeap $\neq \emptyset$ then
        var $\leftarrow$ random element from vsidsHeap;
        remove var from vsidsHeap;
    fi
    while $\neg$ end $\wedge$ (var $\equiv$ undefined $\vee$ var is assigned) do
        if vsidsHeap $\equiv \emptyset$ then
            end $\leftarrow \top$;
        else
            var $\leftarrow$ top element from vsidsHeap;
            remove var from vsidsHeap;
        fi
    done
    if $\neg$end then
        level $\leftarrow$ level+1;
        Assign var with polarity[var] as sign and with no reason (Alg. 2.5, page 23) ;
    fi
end
```

Occurence list Initially, in classic DLL solvers, this was done by checking every clause containing the opposite of the propagated literal. To do this, each literal has a list containing every clause using it. To check those clauses, the search process has to through every literal of the clause to see if there were more than 1 remaining unassigned literal. Therefore, each literal l needs an occurence list: a list containing every clause using l. This implies that a clause C containing n literals is present in n occurence lists. An example of occurence list is depicted in Figure 2.6(a)

Trie data structure The authors of [Zhang and Stickel, 2000] proposed the trie data structure in order to do this more efficiently. A trie data structure is a data structure that does not keep the list of every clause using a given literal. The idea is that using trie data structure, instead of having the full knowledge of a clause (satisfied, unsatisfied, unary, ...), the algorithm accepts to loose the knowledge about the satisfiability of the clause, but not on the unsatisfiability nor on a clause becoming unary. To do this, only two literals are needed. The condition on those two literal (h and t) is that they either have to be unassigned or assigned to $\top$. If one of them is assigned to $\bot$, a new one is needed. If no new one is found, the clause is unary if the other literal is not assigned or the clause is unsatisfied if both are assigned to $\bot$.

In their implementation, the authors of [Zhang and Stickel, 2000] are using two indices i_h and i_t to find the literals h and t. Initialy, h is the first literal of the clause and therefore $i_h = 0$, and t is the last literal of the clause and therefore $i_t = n - 1$. When h is assigned to $\bot$, the algorithm increments i_h until the i_h-th literal is unassigned or assigned to $\top$. When t is assigned to $\bot$, the algorithm decrement i_t until the i_t-th literal is unassigned or assigned to $\top$. If $i_t = i_h$, the clause is unary and if $i_t > i_h$, the clause is unassigned.

Watched Literals Lazy data structures are an improvement of trie data structures, by [Moskewicz et al., 2001], with the watched literals structure. Instead of having two indices representing the position of the literals h and t, those will be moved around such that h is the first literal and t is the second. This simple idea has great benefits during backtracking. Whereas in the implementation of [Zhang and Stickel, 2000] need to modify the value of i_h and i_t when backtracking, this new structure does not, making the backtrack much faster.

The lazy data structure's principle provide an invariant for unsatisfied clauses. Let $\mathcal{L}$ be the literals appearing in a clause C, the literals candidates are a subset $\mathcal{W} \subseteq \mathcal{L}$. $\mathcal{W}$ contains at least one unassigned literal or a literal assigned to $\top$. If this invariant does not hold, a conflict has appeared as the clause is unsatisfied. The question that holds is: regarding the operations the algorithm has to perform (propagation, backtrack, conflict detection) what would the preferable size of $\mathcal{W}$ be? Whenever a watched literal is assigned to $\bot$ and no literal of the clause has been assigned to $\top$, a new watch needs to be looked up, no matter the size of $\mathcal{W}$. If we use only 1 watched literal, whenever a new watch is needed, the algorithm looks up for a new watch and also checks that no propagation is needed. But using only 1 watched literal leads to miss some conflicts (see Example 2.9). If 2 literals are watched in the clause and a new watch is needed, we might be incapable to find one that has not yet been assigned or assigned to $\bot$. In such case, the second watched literal is the literal to propagate. If we use more than 2 literals, special cases are needed for binary clauses and it also increases the number of clause being watched per literal. For those reasons, using 2 literals seems like a reasonable choice.

Example 2.9: 1 watched literal Let Σ be $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ with $C_1 = a \vee b \vee c$, $C_2 = \neg a \vee b \vee c$, and the variables a, b and c do not appear in C_i, $2 \leq i \leq n$. The watcher of a clause is its first literal. Thus, we have for each literal a set of clauses being watched by said literal.

$$\begin{array}{llll} a \rightarrow & \{C_1\} & \neg a \rightarrow & \{C_2\} \\ b \rightarrow & \emptyset & \neg b \rightarrow & \emptyset \\ c \rightarrow & \emptyset & \neg c \rightarrow & \emptyset \\ \ldots & & & \end{array}$$

If the first decision of the solver is to assign $b = \bot$, it does not go through every clause containing b since it uses lazy data structure. If the second decision is to assign $c = \bot$, we can see that a conflict rises through the clauses C_1 and C_2. However, the solver is not able to recognise it since it does not go through those clauses. Therefore, as long as the conflict is not detected, the solver tries to extend a partial assignation that can not be part of a model.

End of Example 2.9

For each literal `l`, a list `watch[l]` is made referencing every clause watching this literal. Different structures have been proposed and implemented but there is mainly two different variations in the literature [Biere, 2008, Eén and Sörensson, 2004] Both provide access to the clauses watching a given literal l through a vector indexed by said literal. The first one, depicted in Figure 2.6(b) uses an array containing a reference to clauses using the indexing literal. It can also contain a second literal for each referenced clauses. This implementation provides fast iteration as the reference to the clauses are contiguous. The insertion of a clause can be done in amortized constant time by adding the clause at the end of the vector. However, removing a clause without changing the order is done in linear time to the number of clauses watching the literal as they have to be shifted to avoid an "empty" cell.

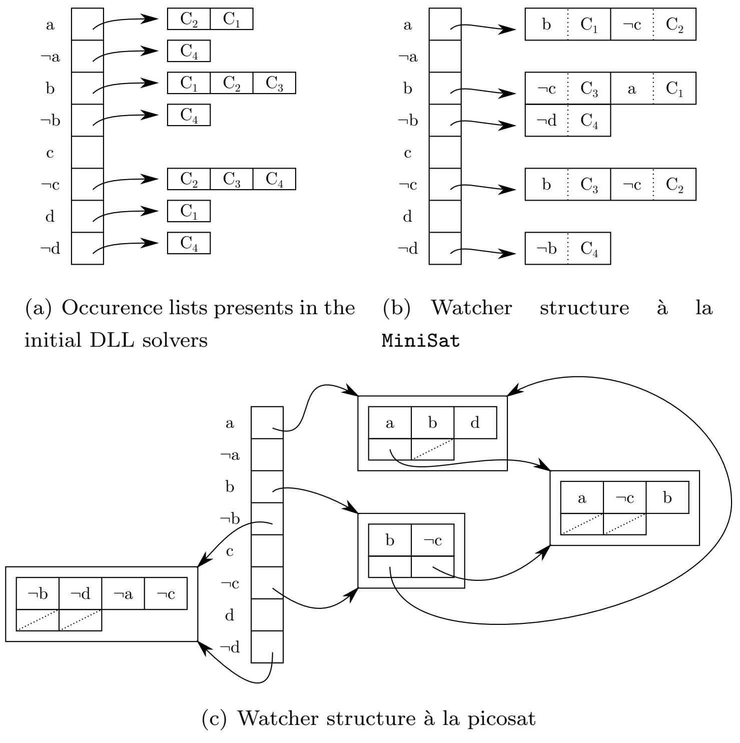
The second version, depicted in Figure 2.6(c) uses a linked list where the cell elements are the clause themselves. This means that in order to look at any information of the clause, a pointer dereference is needed. For this implementation, iterating through the clauses may be slower as in order to obtain any information to the next clause, we must 'follow' a reference. Adding an element to the end of the list can be done in constant time using a dedicated pointer to the last element of the list. Finally, the removal of a clause can be done in constant time as we want to remove a clause when we are iterating through the list, therefore having a pointer to the previous element. By modifying the 'next' pointer of the previous element, we can effectively remove the clause without changing the order in constant time.

Further information about watchers, or occurrence lists, can be found in [Biere, 2008].

The propagation algorithm is depicted in Algorithm 2.7. Whenever a truth assignment has been made, we can consider it as a literal l set to $\top$. Therefore, the opposite literal $\neg l$ can not watch clauses any more, forcing us to find new watches for those clauses. When looking for a new watch, two different outcomes can appear. First, we find a literal that can be watched. In this case, the invariant of watched clause stays true (line 15). The second appears if no other literal can be watched. In that case we either have found a literal that must be propagated

Algorithm 2.7: CDCL:propagation

```
Input : trail: the array containing every literal that must be/have been propagated
Input : top: integer representing the position of the last literal in trail +1
Input : head: integer representing the position in trail of the first literal that has not
        been propagated yet
Input : watch: array providing access for every literal to the list of clauses being
        watched
Input : assign: array providing for each variable its truth value or undefined
Data: lit, lTmp, nl: literals
Data: w: a cell of a linked list containing as data a clause
Data: c: a clause
begin
    while head $<$ top do
        lit $\leftarrow$ trail[head] ;
        w $\leftarrow$ watch[lit];
        while w $\neq$ nil do
            c $\leftarrow$ w.data;
            lTmp $\leftarrow$ other watch in c ;
            nl $\leftarrow$ find new watch in c ;
            if nl $\equiv$ undefined $\wedge$ assign[lTmp] $\equiv$ undefined then
                Assign lTmp because of c (Alg. 2.5, page 23) ;
            elif nl $\equiv$ undefined $\wedge$ assign[lTmp] $\equiv \perp$ then
                unsatCls $\leftarrow$ c ;
                top $\leftarrow$ head;
            elif nl $\neq$ undefined then
                remove w from watch[lit];
                add c in watch[nl];
            fi
            w $\leftarrow$ w.next ;
        done
        head $\leftarrow$ head +1;
    done
end
```



(a) Occurence lists presents in the initial DLL solvers

(b) Watcher structure à la `MiniSat`

(c) Watcher structure à la picosat

Figure 2.6: watcher structure for $\Sigma = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = a \vee b \vee d$, $C_2 = a \vee \neg c \vee b$, $C_3 = b \vee \neg c$ and $C_4 = \neg b \vee \neg d \vee \neg a \vee \neg c$

in order for the clause to be satisfied (line 10). Or, every literal appearing in the clause are evaluated to $\bot$, leaving an unsatisfied conflicting clause (line 12).

The assignation algorithm is quite simple. First, the falsified literal needing propagation is added to the `trail` and the value of `top` is increased correspondingly. After that, the assigned value of the variable `v` is set in `assign` and the reason of the assignation, if any, is put in `reason`. Finally, the level at which the variable was assigned is saved in `varLevel`.

Example 2.10: Propagation example Let $a, b, c, d, e, f, g, h, i$ be variables and $\mathcal{C}$ the following set of clauses:

$$\begin{array}{llll} C_1 = & \neg a \vee b & C_2 = & \neg a \vee d, \\ C_3 = & \neg e \vee g & C_4 = & \neg g \vee f \vee \neg b, \\ C_5 = & \neg f \vee i & C_6 = & \neg f \vee \neg i \vee \neg d, \\ C_7 = & \neg h \vee c & & \end{array}$$

And the list of watches is the following:

$\texttt{watch}[a]$	$= nil$	$\texttt{watch}[\neg a]$	$= C_1, C_2$
$\texttt{watch}[b]$	$= C_1$	$\texttt{watch}[\neg b]$	$= nil$
$\texttt{watch}[c]$	$= C_7$	$\texttt{watch}[\neg c]$	$= nil$
$\texttt{watch}[d]$	$= C_2$	$\texttt{watch}[\neg d]$	$= nil$
$\texttt{watch}[e]$	$= nil$	$\texttt{watch}[\neg e]$	$= C_3$
$\texttt{watch}[f]$	$= C_4$	$\texttt{watch}[\neg f]$	$= C_5, C_6$
$\texttt{watch}[g]$	$= C_3$	$\texttt{watch}[\neg g]$	$= C_4$
$\texttt{watch}[h]$	$= nil$	$\texttt{watch}[\neg h]$	$= C_7$
$\texttt{watch}[i]$	$= C_5$	$\texttt{watch}[\neg i]$	$= C_6$

Suppose that the first decision variable chosen is a. $\neg a$ is added to the trail, the level of variable a is 1 and there is no clause that leads to this choice. These information are depicted using the following schema $\langle x@_y r\rangle$, where x is the literal present in `trail`, y the level of the variable and r is the clause that leads to the propagation, or nil if there was none. In our example, we have $\langle \neg a@_1 nil\rangle$.

The propagation algorithm examines clauses watched by $\neg a$ at line 3. Those are C_1 and C_2. For C_1, the other watch obtained at line 7 is b. As both $\neg a$ and b are watched, no new watch can be found at line 8. b not being assigned, it is assigned at line 10, obtaining $\langle \neg a@_1 nil, \neg b@_1 C_1\rangle$. After that, the next clause considered (line 18) is C_2. As b for C_1, d is assigned because of C_2.

As every clause watched by $\neg a$ have been processed, the next literal to propagate is examined: $\neg b$. However, no clause is watched by it. Finally, $\neg d$ is processed but does not watch any clause either.

End of Example 2.10

2.2.3 Conflict analysis

The main idea behind the conflict analysis in CDCL solver is to create a clause C helpful for the search by providing a way to avoid the same conflict after a backtrack or restart.

Simple implementation

One simple way to create a clause that provides such benefits is to use the negation of the decision literals. However, this simple method have a few drawbacks. The first one is the fact that a conflict is not necessary dependent of every choice made since the first decision level. The second drawback is that the generated clause does not provide useful information for the backtracking process as the latest decision is the only decision that can be reverted.

Example 2.11: Drawback of the simple implementation Let the set of variables be $x_i, 0 \leq i < 10$, set of clauses be $\mathcal{C} = \{C_1 = \bigvee_{i=0}^{10} x_i,\ C_2 = \neg x_4 \vee \neg x_5 \vee \neg x_6,\ C_3 = \neg x_4 \vee \neg x_5 \vee x_6\}$, and the current interpretation be $x_j@j, 0 \leq j < 6$. Given the interpretation, a conflict occurs as C_2 implies $\neg x_6$ and C_3 implies x_6. Using the simple implementation, a new clause C_4 is created of the form $\bigvee_{i=0}^{6} \neg x_i$. However, the generated clause C_4 is too long as $\neg x_4 \vee \neg x_5$ would have been sufficient.

Now, let the set of variables be x_0, x_1, x_2, x_3, the set of clauses be $\mathcal{C} = \{C_1 \equiv \neg x_0 \vee \neg x2 \vee \neg x_3, C_2 \equiv \neg x_2 \vee x_3\}$, and the current interpretation be $x_0@0, x_1@1, x_2@2$. Given the interpretation, a conflict occurs as C_1 implies $\neg x_3$ and C_2 implies x_3. Using the simple implementation, a new clause $C_3 \equiv \neg x_0 \vee \neg x_1 \vee \neg x_2$ is generated. However, if the clause $C_3' = \neg x_0 \vee \neg x_2$ would have been generated, the solving process could have reverted the decision x_3 and x_1 as none of those decision variable are present in C_3'

End of Example 2.11

Unique implication point

Let us first define the concept of dominator. This concept is based on dominator in the implication graph: let x_a and x_b be two vertices in an implication graph $\mathcal{G}$, the assignation level of x_a being d and x_c the vertex representing the decision variable of level d. x_a is called the dominator of x_b if every path from x_c to x_b goes through x_a. When a conflict rises, the vertices x and $\neg x$ appear in the implication graph at level d, we can compute the set $\mathcal{D}$ of dominators on the vertices x and $\neg x$. Such set can not be empty as it contains at least the last decision variable. The elements of this set are called unique implication point (or UIP). Those were used in [Marques-Silva and Sakallah, 1997]. By using the resolution operator on the conflict vertices and their antecedents, it is possible to obtain a clause containing only one literal, a unique implication point. As there might be multiple UIP, [Zhang et al., 2001] pointed out that CDCL solver may stop at the first UIP encountered starting from the conflict. Moreover, in [Audemard et al., 2008], the authors prove that the first UIP provides the highest level to backjump to.

Example 2.12: Unique implication point Let a,b,c,d,e,f,g,h and i be variables and Σ the following set of clauses:

$$\begin{array}{llll} C_1 = & \neg a \vee b & C_2 = & \neg c \vee d, \\ C_3 = & \neg b \vee \neg d \vee e & C_4 = & \neg e \vee g, \\ C_5 = & \neg g \vee f & C_6 = & \neg g \vee \neg f \vee \neg h, \\ C_7 = & \neg h \vee i & C_8 = & \neg h \vee \neg i \end{array}$$

and the trail be $\langle \neg a@_1 nil$, $\neg b@_1 C_1$, $\neg c@_2 nil$, $\neg d@_2 C_2$, $\neg e@_2 C_3$, $\neg g@_2 C_4$, $\neg f@_2 C_5$, $\neg h@_2 C_6$, $\neg i@_2 C_7\rangle$ and the conflicting clause being C_8 on the literal $\neg i$. The related implication graph is depicted in Figure 2.7. In this situation, there are multiple UIP: c, d, e and h. As h is the nearest UIP from the conflict, it is called the first UIP.

End of Example 2.12

Actual implementation

With the conflict analysis, we want to be able to deduce when was made the affectation leading to the current conflict. To do this, we generate a clause from the current conflict using the

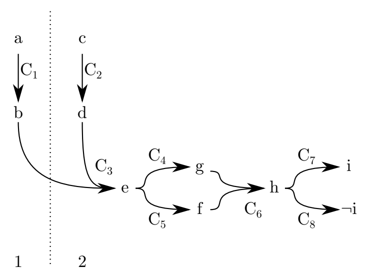


Figure 2.7: Implication graph depicting UIP c, d, e and h

Algorithm 2.8: CDCL:conflict:analysis

```
Input : varLevel: the level of each variable if the variable is assigned
Data: i: the index to the last unvisited of trail
Data: seen: a Boolean array over each variable
Data: n: the number of variables of the current level on which a resolution needs to be
      performed
Data: v: a variable
Data: c: a clause
Data: lTmp: a literal
1  begin
2      n ← 0, c ← unsatCls, i ← top ;
3      initialize every element of seen to ⊥;
4      repeat
5          foreach lTmp in c do
6              v ← variable of lTmp;
7              update VSIDS for v (Alg. 2.4, page 21);
8              if ¬ seen[v] ∧ varLevel[v] > 0 then
9                  seen[v] ← ⊤;
10                 if varLevel[v] ≡ level then
11                     n←n+1;
12                 else
13                     add lTmp in confCls;
14                 fi
15             fi
16         done
17         repeat decrement i until seen[variable of trail[i]]≡ ⊤;
18         lTmp ← trail[i], v ← variable of lTmp;
19         c ← reason[v], i ← i−1, n ← n−1;
20     until n ≡ 0 ;
21     add ¬lTmp in confCls;
22 end
```

resolution rule (defined in Equation 1.3, page 6) that holds only one variable v of the current level. Using this, the solver can go back until v was unassigned and avoid the conflict.

The algorithm analysing the conflict is depicted in Algorithm 2.8. The idea behind this algorithm is the following: from the conflicting clause, we create a new clause using the resolution operator $\otimes_{\mathcal{R}}$. The resolution is applied on literals of the current level as our goal is to have only one of those literals.

Example 2.13: Conflict clause analysis Let a, b, c, d and e be variables, $C_1 = a \vee d$, $C_2 = b \vee \neg d$, $C_3 = \neg c \vee a \vee \neg b$, $C_4 = \neg e \vee \neg c \vee \neg a$. Let the assignment be $\{e@_1 nil,\ c@_2 nil,\ \neg a@_2 C_4,\ d@_2 C_1,\ \neg b@_2 C_3\}$. This assignment leads to a conflict as $\neg a$ implies d and $\neg b$ implies $\neg d$. Therefore, the conflicting clause is C_2. This conflicting clause is used as starting point for the conflict analysis and resolutions are performed until there is only one variable left from the current level, equal to 2.

$$\begin{array}{rclcr} R_1 & = & C_2 \otimes_{\mathcal{R}} C_1 & = & a \vee b \\ R_2 & = & R_1 \otimes_{\mathcal{R}} C_3 & = & a \vee \neg c \\ R_3 & = & R_2 \otimes_{\mathcal{R}} C_3 & = & \neg e \vee \neg c \end{array}$$

End of Example 2.13

When parsing the clauses used to create our resolvent, two different cases can appear. On one hand, if the literal is from a previous level, we may add it in our resolvent that we are constructing. On the other hand, if the literal is from the current level, we have to put it aside in a set called $\mathcal{S}$. This set must contain only one element. If not, we remove the literal that has been propagated last in chronological order from the set and continue to analyze by making a resolution uppon said literal.

During Algorithm 2.8, we have to make sure to treat each variable at most once as we should not add twice the same literal in the resolvent. To ensure this, we use the Boolean array `seen` such that for a given variable `v`, `seen`[`v`] is true when we have processed `v`. Furthermore, $\mathcal{S}$ can be implicit as we only need a value `n` from which we can derive its size. The content from $\mathcal{S}$ can be deduced from the content of `trail`. Finally, the set of literals from previous level that appears in our resolvent is stored in `confCls`.

The following invariants hold for the loop at line 4.

- $\mathcal{I}_1$: `n` +1 is the number of literals of current level that should appear in the result of the resolutions.
- $\mathcal{I}_2$: `confCls` holds the literals of previous level that appear in the result of the resolutions.
- $\mathcal{I}_3$: `lTmp` is a literal of current level.
- $\mathcal{I}_4$: If $\neg l_i$ is present in trail at position p_i, $p_i <$ `i` and l_i is a literal of current level and `seen`[$variable\ of\ l_i$] is $\top$, then l_i is a literal that should be in the resolvent. There are `n` such literals in `trail`.
- $\mathcal{I}_5$: Since there are `n` +1 missing literals in the resolvent, the last one is the opposite of `lTmp`.

When the condition at line 20 is false, there is only one missing literal since `n` $= 0$. The algorithm is composed of two phases. The first starts with the loop at line 4 and adds to `confCls` the literals of previous level. The second is starting at line 21 and adds a literal of the current level.

Example 2.14: conflict analysis Let $a, b, c, d, e, f, g, h, i$ be variables and $\mathcal{C}$ the following set of clauses:

$$\begin{aligned} C_1 &= \neg a \vee b & C_2 &= \neg a \vee d, \\ C_3 &= \neg e \vee g & C_4 &= \neg g \vee \neg b \vee f, \\ C_5 &= \neg f \vee i & C_6 &= \neg d \vee \neg f \vee \neg i, \\ C_7 &= \neg h \vee c \end{aligned}$$

Let the trail be $\langle \neg a@_1 nil$, $\neg b@_1 C_1$, $\neg d@_1 C_2$, $\neg c@_2 nil$, $\neg e@_3 nil$, $\neg g@_3 C_3$, $\neg f@_3 C_4$, $\neg i@_1 C_5\rangle$ and C_6 the unsatisfied clause. The corresponding implication graph is depicted in Figure 2.8.

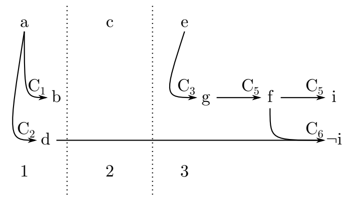


Figure 2.8: Implication graph for the conflict in Example 2.14

`c` is initialized with C_6 and `i` with the value 8. The first loop goes through the literals of C_6. Both $\neg i$ and $\neg f$ are from the current level. Therefore, `n` is incremented twice and `seen` for those variables is set to $\top$. $\neg d$ is not from the current level, therefore it is added in `confCls`. On line 17, `i` only needs to be decremented once to select the variable i, and `c` is C_5. At the end of the loop, `n` now equals 1.

For the second pass through the loop, both variables i and f were already seen and there are no literal of previous level. No change is made neither to `confCls` nor to `n`. On line 17, `i` is decremented once and f is selected. As `n` is decremented, it is now equals to 0 and therefore, the condition on line 20 is $\top$. The last line of the algorithm adds the opposite of the last selected literal in `c` making it the clause $C_8 = \neg f \vee \neg d$. As expected, C_8 has only one literal of the current level.

End of Example 2.14

Clause simplification

When analysing the source of the conflict, the generated clause might be quite long. Different techniques have been proposed to reduce the size of such generated clauses e.g. : [Sörensson and Eén, 2005], [Sörensson and Biere, 2009]. They both use the self-subsuming scheme (see page 6). Let us recall that a self-subsuming clause is a clause obtained by resolution of two clauses and the resolvent subsumes one of the initial clauses.

For each literal of the generated clause, we have to check if it can be safely removed. To remove l safely, every literal of the reason of l must be contained in the generated clause or it must be itself safely removed.

Throughout Algorithm 2.9, we consider that if `seen[v]` is assigned to $\top$, it means that at some point of the creation of the generated clause, we removed `v` it through the operator $\otimes_{\mathcal{R}}$. The recursivity is implemented using a stack `s` where on top of it is the literal that we are trying to check if it can be safely removed.

Example 2.15: simplification example Let us consider the following clauses:

$$\begin{aligned} C_1 &= \neg b \vee d & C_2 &= \neg d \vee e, \\ C_3 &= \neg d \vee \neg e \vee i & C_4 &= \neg k \vee \neg f \\ C_5 &= \neg d \vee \neg e \vee \neg i \vee \neg k \vee f \end{aligned}$$

and the trail $\langle \neg b@_1 nil,\ \neg d@_1 C_1,\ \neg e@_1 C_2,\ \neg i@_1 C_3,\ a@_2 nil,\ \neg k@_3 nil,\ \neg f@_3 C_5 \rangle$. The corresponding implication graph is depicted in Figure 2.9. The conflicting clause is C_6 and the generated clause is $C_7 = C_6 \otimes_{\mathcal{R}} C_5 = \neg d \vee \neg e \vee \neg i \vee \neg k$. The simplification algorithm tries to remove each of the literal of C_7. First, $\neg d$ was assigned to $\bot$ because of C_1. Therefore, the algorithm examines the literals $\neg b$ and d. As $\neg b$ has no reason, `b` $\leftarrow \bot$ and the second literal of C_1 is not checked. As a result, $\neg d$ can not be removed from C_7.

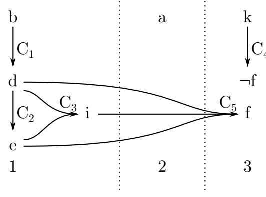


Figure 2.9: Implication graph for the conflict in Example 2.15

Next, we consider the second literal of C_7, namely $\neg e$. It was assigned because of C_2. As the variable d has already been seen, we can deduce that $\neg e$ can be removed safely from C_7. We obtain:

$$\begin{aligned} C_7' &= C_7 \otimes_{\mathcal{R}} C_2 \\ &= \neg d \vee \neg i \vee \neg k \end{aligned}$$

The next literal to consider from the generated clause is $\neg i$, assigned because of C_3. From that clause, every literal has already been seen, stating that $\neg i$ can also be removed. We obtain:

$$\begin{aligned} C_7' &= (C_3 \otimes_{\mathcal{R}} C_2) \otimes_{\mathcal{R}} C_7' \\ &= (\neg d \vee \neg i) \otimes_{\mathcal{R}} C_7' \\ &= \neg d \vee \neg k \end{aligned}$$

Finally, the last literal to consider, $\neg k$ has no reason, therefore can not be removed.

Algorithm 2.9: CDCL:conflict:simplification

```
Data: p: The literal to check if it can be safely removed
Data: s: a stack of literals
Data: lit: a literal
Data: lTmp: a literal
Data: v: a variable
Data: c: a clause
Data: b: the Boolean stating that p can be safely removed
begin
    foreach p in confCls do
        add p in s ;
        b ← $\top$ ;
        while s is not empty $\wedge$ b $\equiv \top$ do
            lit ← top element of s;
            remove top element from s;
            c ← reason[variable of lit];
            if c $\neq$ nil then
                foreach lTmp in c do
                    v ← variable of lTmp;
                    if $\neg$ seen[v] $\wedge$ level of v $> 0 \wedge$ b $\equiv \top$ then
                        if reason[v] $\neq$ nil then
                            seen[v] ← $\top$;
                            add lTmp in s;
                        else
                            revert modification on seen;
                            clear s;
                            b ← $\perp$ ;
                        fi
                    fi
                done
            else
                b ← $\perp$;
            fi
        done
        if b $\equiv \top$ then
            remove p from confCls;
        fi
    done
end
```

As we can see from our example, the simplification mechanism leads to the simplification of $\neg d \vee \neg e \vee \neg i \vee \neg k$ to $\neg d \vee \neg k$.

End of Example 2.15

2.2.4 Non-chronological backtracking

After each conflict, we must revise the previous choices made to take into account the conflict just discovered. This backjumping (or non-chronological backtracking) has as destination the moment when the new clause should have been used to propagate a literal, effectively avoiding the conflict.

Revising previous choices

When reverting previous choices, we need to update the following structures: `assign`, `varLevel`, `reason` and `trail` related information. For those structures, we have to modify their value corresponding to variables at a level higher than the destination level `dstLevel`. Those variables are at the top of the trail and therefore, we may traverse it backwards until the variable at the top of the trail is from the destination level. As for the occurrence list, no change is needed as the watched literals may only become unassigned and unassigned literals are still valid.

Algorithm 2.10: CDCL:backtracking

```
Input : varLevel: the level of each variable if the variable is assigned
Input : dstLevel: the level we want to backtrack to
Data: v: a variable
Data: lTmp: a literal
begin
    lTmp ← trail[top−1], v ← variable of lTmp;
    while varLevel[v] > dstLevel do
        polarity[v] ← assign[v];
        assign[v] ← undefined;
        varLevel[v] ← $\infty$, reason[v] ← nil ;
        top ← top−1, lTmp ← trail[top−1], v ← variable of lTmp;
    done
end
```

Example 2.16: backjumping Let Σ be the following set of clauses:

$$
\begin{array}{llll}
C_1 = & \neg a \vee b & C_2 = & \neg c \vee \neg b \vee d, \\
C_3 = & \neg d \vee e & C_4 = & \neg f \vee \neg d \vee g \\
C_5 = & \neg g \vee \neg e \vee h & C_6 = & \neg i \vee j \\
C_7 = & \neg j \vee k & C_8 = & \neg b \vee \neg e \vee \neg j \vee \neg k
\end{array}
$$

the trail be $\langle \neg a@_1 nil,\ \neg b@_1 C_1,\ \neg c@_2 nil,\ \neg d@_2 C_2,\ \neg e@_2 C_3,\ \neg f@_3 nil,\ \neg g@_3 C_4,\ \neg h@_3 C_5,\ \neg i@_4 nil,\ \neg j@_4 C_6,\ \neg k@_4 C_7\rangle$ and the conflicting clause C_8. The current

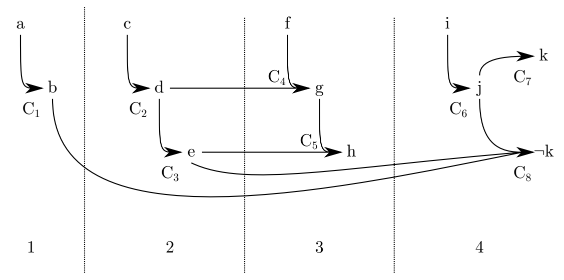


Figure 2.10: Implication graph for the conflict in Example 2.16

state of the `watch` arrays is depicted at Table 2.2 and the related implication graph at Figure 2.10.

`watch`$[a]$	$= nil$	`watch`$[\neg a]$	$= C_1$
`watch`$[b]$	$= C_1$	`watch`$[\neg b]$	$= nil$
`watch`$[c]$	$= nil$	`watch`$[\neg c]$	$= C_2$
`watch`$[d]$	$= C_2$	`watch`$[\neg d]$	$= C_3$
`watch`$[e]$	$= C_3$	`watch`$[\neg e]$	$= nil$
`watch`$[f]$	$= nil$	`watch`$[\neg f]$	$= C_4$
`watch`$[g]$	$= C_4$	`watch`$[\neg g]$	$= C_5$
`watch`$[h]$	$= C_5$	`watch`$[\neg h]$	$= nil$
`watch`$[i]$	$= nil$	`watch`$[\neg i]$	$= C_6$
`watch`$[j]$	$= C_6$	`watch`$[\neg j]$	$= C_7, C_8$
`watch`$[k]$	$= C_7$	`watch`$[\neg k]$	$= C_8$

Table 2.2: Watchers for the Example 2.16

From this state, the solver learns the clause $C_9 = \neg b \vee \neg e \vee \neg j$. We can see that this clause uses only one literal of the current level: $\neg j$. The level to backjump to is the second highest level of the clause (the first being the literal of the current level). For C_9, the second highest level is 2: the level of $\neg e$. The backtracking algorithm will therefore cancel the decision taken after the second level and their propagated literals. The trail will become $\langle \neg a@_1 nil,\ \neg b@_1 C_1,\ \neg c@_2 nil,\ \neg d@_2 C_2 \rangle$. We can see that with the generated clause, we were able to 'jump' from the level 4 to level 2, effectively skipping level 3. Concerning the `watch`, since a watched literal may be unassigned variable, no change is needed to the structure after the backtrack.

Now, let us imagine that $C_8 = \neg j \vee \neg k$. As result, the learnt clause is now $C_9 = \neg j$. As the clause is unary, there is no second highest level and therefore, the backjump performed has as destination, the level 0.

End of Example 2.16

Processing of new clause

Once a clause is generated, it is used at least once to revise a previous choice (see Section 2.2.4). That generated clause must be added in a dedicated data structure, and the watches specified. The algorithm is depicted in Algorithm 2.11.

Algorithm 2.11: CDCL:conflict:addingClause

```
  Input  : c: the clause that needs to be added
  Input  : generatedDB: the set of generated clauses
  Data: lTmp1, lTmp2: literals
1 begin
2     add c to generatedDB;
3     let lTmp1 be the first watched literal ;
4     let lTmp2 be the second watched literal ;
5     add c to watch[lTmp1];
6     add c to watch[lTmp2];
7 end
```

The main question that must be resolved is the choice of literals as watches. When generating this clause, we made sure that there was only one variable of the current level to help during the backtracking. If the generated clause is of size greater or equal than one, a second watch must also be used. The value used is the second most recent propagated literal.

2.2.5 Restart strategies

During the search, the process can be stuck at a given depth as its current search space does not provide enough information. Of course, as the algorithm is complete, one could let the procedure continue. However, restarting from the initial level i.e. performing a backtrack to level 0, can provide a great boost in performances [Gomes et al., 1998]. However, this idea poses multiple questions: what should the frequency be? And using which time unit? Concerning the time unit, the number of conflicts can be used. As for the frequency, multiple approaches exist as we can see in the following.

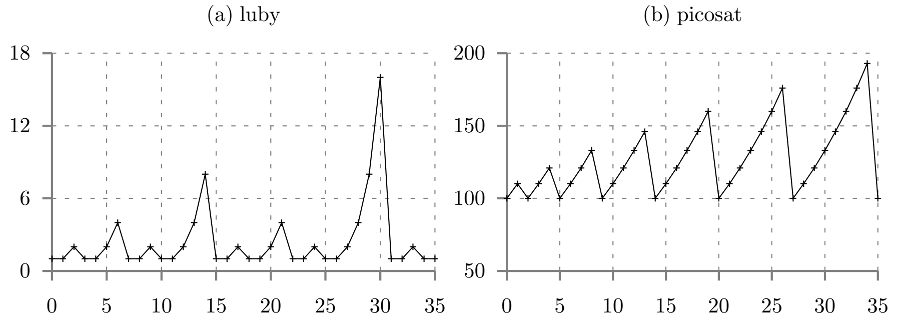


Figure 2.11: Comparison of restart strategies. y coordinates give the number of conflicts between restart x and $x - 1$.

Luby suite

The paper by [Luby et al., 1993a] provides information about restart strategies for Las Vegas algorithms. Such algorithms are guaranteed to provide a correct answer, but the running time is unknown. We can clearly see that the CDCL algorithm fits this description.

Let $T_{CDCL}(x)$ be the running time for the CDCL algorithm for any instance x. This can be seen as a random variable whose distribution is unknown. For such cases, the suite $S = \{t_0, t_1, \ldots\}$ shown at Equation 2.1 is proven to provide the best performances up to a constant factor. The obtained values are shown in Figure 2.11 (a).

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^k - 1 \leq i < 2^k - 1 \end{cases} \tag{2.1}$$

In solvers, the Luby suite is generally multiplied with an constant value to have greater result value. If the value of the Luby suite was not multiplied, the frequency of restarts would be too high (30 restarts after only 64 conflicts).

Picosat restarts

Another possible suite is the one implemented in [Biere, 2008], inspired by the Luby suite. It consists of a succession of exponential curves, each restarted after they outreach a given peak. The formal equation is provided in Equation 2.2 The suite $S = \{t_0, t_1, \ldots\}$ needs different values to be computed: t_0, e and h_0. The obtained values with $t_0 = 100$, $e = 1.1$ and $h_0 = t_0$ is depicted in Figure 2.11 (b).

$$\langle t_i, h_i \rangle = \begin{cases} \langle t_0, & h_{i-1} \times e \rangle, & \text{if } t_{i-1} \geq h_{i-1} \\ \langle t_{i-1} \times e, & h_{i-1} \rangle, & \text{if } t_{i-1} < h_{i-1} \end{cases} \tag{2.2}$$

Average literal block distance

The two previous methods are using a fixed restart policy. No matter what is happening during the search, the restart strategy is fixed across different instances. In the paper [Audemard and Simon, 2009a], the authors propose a metric, called *literal block distance*(or *lbd*) for each generated clause. The value of this metric is obtained by partitioning literals according to their assignation level and then computing the number of partitions. Through experiments, they discovered that clauses with a small *lbd* value tend to propagate more. Therefore, they designed a restart policy to promote such clauses.

When the metric is obtained, two averages are computed. The first one $\overline{L_{100}}$ is the average *lbd* on the last 100 generated clauses. The second $\overline{L}$ is the average *lbd* for every generated clause. Finally, whenever Equation 2.3 evaluates to $\top$, a restart is performed. K is a constant value that needs to be determined through experimentation but 0.8 seems to provide good results. N_{conf} is the number of conflicts since the last restart.

$$N_{conf} > 100 \wedge \overline{L_{100}} \times K > \overline{L} \tag{2.3}$$

The idea behind this system is that whenever $\overline{L}$ might be increasing too fast, a restart is performed to return to "the good search space". This technique was shown to be very effective on *unsatisfiable* instances but the gain was smaller for *satisfiable* instances. To improve this, changes were proposed in [Audemard and Simon, 2012]. Their idea is to delay restarts when the number of assigned literals is significantly greater than what they usually have. Two new values are defined: $\overline{A_{5000}}$ and R. R is a constant value and $\overline{A_{5000}}$ is the average number of assignations over the last 5000 conflicts. When a conflict rises, if the current number of assignations is greater than $\overline{A_{5000}} \times R$ then N_{conf} is set to zero.

2.2.6 Clause database management

As we have seen, a new clause is added at each conflict. Those clauses have a direct impact on the search speed as we may have to check them during the propagation algorithm (page 28). But on the same time, they are useful as they provide new information from the solver perspective. Therefore, a compromise on their number needs to be found. In an ideal world, we need to remove the clauses that are subsumed, and those which are not needed any more in the search process. Unfortunately, those elements need heavy computation and are almost never explicitly used in the latest CDCL solvers. Therefore, different strategies have been proposed through heuristic to determine which clauses are "the good ones" and which are not. The next paragraphs describe some of them as well as the frequency those strategies are applied.

Clause activity

A common technique to remove useless clauses is to compute the activity. Such activity is computed using the same idea as the VSIDS (see page 19). A score is given for each clause. Each time a clause C is used in the resolution to generate the learnt clause, the activity of C is incremented by a given factor. Once the learnt clause is generated the increment factor is multiplied by a constant. Later, when some clauses need to be removed, we may remove those with the lowest activity value.

Aggressive deletion

As the amount of clauses has a significant effect on the number of propagations per second, the authors of [Audemard and Simon, 2009b] proposed to use their newly defined metric *lbd* to sort the clauses. Next, over the n clauses they have, they keep only $n/2$ of the clauses with the smallest *lbd*. The frequency of the application of the strategy is given in number of conflicts using the formula defined at Equation 2.4. The initial value in the `glucose` 1 implementation was $t_0 = 20000$.

$$t_{i+1} = 500 \times (i+1) + t_i \tag{2.4}$$

Freeze

First described in the paper [Audemard et al., 2011a], this technique relies on the following idea: removing a clause is somewhat too definitive as a clause might not have been used a lot at the moment, but could be useful later. From this idea, came the schema shown in Figure 2.12. Learnt clauses are divided in three sets: the active ones, the inactives ones and the deleted. Active clauses are present in the `watch` data-structure in order to guide the search. Inactive clauses, or frozen clauses, are removed from the `watch` data-structure, they can not guide the search as they can not be used for propagation. Deleted clauses are clauses that were completely wiped off the memory of the solver. The propagation algorithm uses only active ones.

When generated, clauses are active. Later, they are examinated and their status can change to inactive or they can be deleted. The inactive ones can be re-activated, stay frozen or be deleted. Every non-deleted clause is evaluated in this process according to a certain frequency. Now that the general outline is given, we must determine whenever a clause is a good candidate for the current search space or not, those are the conditions A, B, C and D in Figure 2.12.

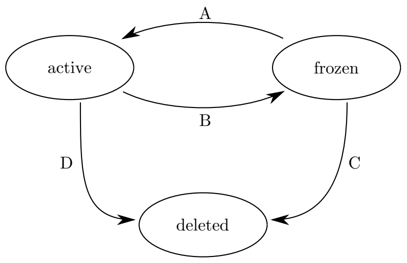


Figure 2.12: Clauses categories for the freezing mechanism

$$d_t = \frac{\mathcal{H}(\texttt{polarity}_t, \texttt{polarity}_{t-1})}{V} \tag{2.5}$$

Let $psm_{\mathcal{P}}(C)$ be the number of variables that appear in the clause C with the same polarity as the one used for the next branching value (see page 22). Let d_t be the deviation after t calls to the clause database management procedure, where the deviation is the normalised Hamming distance between the values of the array `polarity` at t and $t-1$ (see Equation 2.5 where V is the number of variables). Let d_m be the minimum of $d_1, d_2, \ldots, d_t$. Now, if we multiply d_m by the size $|C|$ of the clause C to obtain the value, we know how many variables present in the clause may change their sign until the next clause database management. Therefore, if $psm_{\mathcal{P}}(C)$ is greater than $d_m \times |C|$, we know that this clause could never be used to generate a conflict. Indeed, there are too many literals set to $\top$.

With those new metrics, we can define the conditions A, B, C and D:

A $psm_{\mathcal{P}}(C) < d_m \times |C|$

B $psm_{\mathcal{P}}(C) \geq d_m \times |C|$

C a clause has been frozen for the last n clause database management procedures and has never been activated

D a clause has been active for the last n clause database management procedures and has never been used or its *lbd* value is greater than m

$$t_i = t_{i-1} + \text{inc} \tag{2.6}$$

As for the frequency, the authors of [Audemard et al., 2011a] propose the formula at Equation 2.6 in number of conflicts, with as initial value $t_0 = 500$, inc $= 100$.

2.2.7 Helpful techniques

There are different scenarios concerning SAT solvers. Therefore, different techniques made their way into the algorithms.

Assumptions

One of the usage scenario for SAT solver is the call to different instances and those instances share a high amount of clauses. Let us define Σ_c for the set of common clauses between $\Sigma_1, \ldots, \Sigma_n$. Those $\Sigma_1, \ldots, \Sigma_n$ can be merged into one instance Σ_m using the following principle described in Equation 2.7 where l_i are new variables.

$$\Sigma_m = \Sigma_c \bigwedge_{i=1}^{n} (l_i \wedge \Sigma_i \backslash \Sigma_c) \tag{2.7}$$

Later, when the user wants to solve Σ_i, they add the cube defined in Equation 2.8 to their solver.

$$\neg l_i \bigwedge_{j_{1 \leq j \leq n, i \neq j}} l_j \tag{2.8}$$

Using this technique it is also possible to prefix every clause of an instance, the user can select which clauses to activate.

Later, when clauses using such literals are used in the conflict analysis, the literal may not be removed as there is no appearance of $\neg l_i$. This makes this trick very useful as one can later re-use the learnt clause when new sets of clauses are activated. To do this, the user needs an easy interface to tell the algorithm the assignation for such l_i. This interface is usually called *assumptions*. Assumptions are literals that are supposed to be evaluated to $\top$. When the solver needs to branch, it first checks if an assumption variable is not yet assigned and if so, it assigns the corresponding literal. That way, those literals are not assigned at level 0, making them assumptions instead of facts.

More information about assumptions can be found in [Nadel and Ryvchin, 2012].

Preprocessing

When a user generate its instance, it is sometimes useful to simplify it. Different techniques have been proposed.

SatElite The SatElite pre-processor [Eén and Biere, 2005] uses variable elimination technique, self-subsuming clauses and tautological clause found through unit propagation.

Vivification Proposed in [Piette et al., 2008], this technique aims at simplifying clauses using the following technique. Given a clause $C = x_1 \vee \ldots \vee x_n$, a solver assign progressively every x_i to $\perp$ using the formula $\Sigma' = \Sigma \backslash \{C\}$ Once a x_i has been propagated, different interesting outcomes can appear.

- a conflict is found, meaning that $\Sigma' \vdash_{PU} x_1 \vee \ldots \vee x_i$ and therefore the solver can drop the literals $x_{i+1}, \ldots, x_n$ from C

- $x_j, j > i$ is propagated to $\bot$, meaning that $\Sigma' \vdash_{PU} x_1 \vee \ldots \vee x_i \vee \neg x_j$ and therefore x_j can be dropped from C
- $x_j, j > i$ is propagated to $\top$, meaning that $\Sigma' \vdash_{PU} x_1 \vee \ldots \vee x_i \vee \neg x_j$ and therefore the literals $x_{i+1}, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n$ can be dropped from C

2.2.8 Variables definitions

Here, we recall variables used through the explanation of the CDCL algorithm.

`activity`	array containing the VSIDS score for each variable. 21
`assign`	array providing for each variable its truth value or **undefined**. 23, 28, 29, 37
`confCls`	clause representing a conflict. 18, 32–34, 36
`end`	Boolean, $\top$ if a solution has been found, $\bot$ otherwise. 18, 25
`freq`	frequency on which we want a random variable choice. 25
`generatedDB`	set of generated clauses. 39
`head`	integer representing the position in `trail` of the first literal that has not been propagated yet. 23, 28
`inc`	increment value for the update of VSIDS score of variables. 21
`level`	current assignation level. 23, 25, 32
`polarity`	array containing for each variable the next truth value needed for branching. 22, 25, 37, 42
`reason`	array containing for each variable `v` the clause that forced the value of `v`. 23, 29, 32, 36, 37
`seen`	Boolean array over each variable. 32–36
`top`	integer representing the position of the last literal in `trail` +1. 23, 28, 29, 32, 37
`trail`	array containing every literal that must be/have been propagated. 23, 24, 28–30, 32, 33, 37
`unsatCls`	unsatisfied clause. 18, 28, 32
`varLevel`	array providing the value of the level for each variable. 19, 23, 29, 32, 37
`vsidsHeap`	heap containing the variables, sorted according to their VSIDS score. 19, 21, 25
`watch`	array providing access for every literal to the list of clauses being watched. 27, 28, 30, 38, 39, 41

2.3 Solver example

Since SAT solving is studied from quite some time, many different solvers have been proposed. We present a few of the CDCL solvers.

`MiniSat` Developed by Niklas Eén and Niklas Sörensson, `MiniSat` [Eén and Sörensson, 2004] is probably the most used CDCL SAT solver. It won 3 medals in the SAT competition 2005, 6 in the SAT competition 2007 and has not been developed further since. It was developed to be as concise as possible in order to be easily extendible. Therefore, its code is used by many teams by extending the solver's capabilities or tweaking the heuristics.

`lingeling` Developed by Armin Biere, `lingeling` [Biere, 2014] is the result of years of SAT solver design. It won 4 medals in SAT competition 2011, 5 in the SAT competition 2013 and 9 in the SAT competition 2014.

`sat4j` Developed by Daniel Le Berre and Anne Parrain, `sat4j` [Le Berre and Parrain, 2010] was initially a re-implementation of `MiniSat` using the java programming language. It is used in many practical application including the Eclipse IDE.

`glucose` Developed by Gilles Audemard and Laurent Simon on top of `MiniSat`, `glucose` was proposed in order to implement and evaluate the techniques described in the paper [Audemard and Simon, 2009b]. It won 2 medals in the SAT competition 2009, 3 medals in the SAT competition 2011, 4 medals in the SAT competition 2013 and 2 medals in the SAT competition 2014 `glucose` uses the aggressive clause deletion strategy (page 41) combined with the average *lbd* restart policy (see page 40).

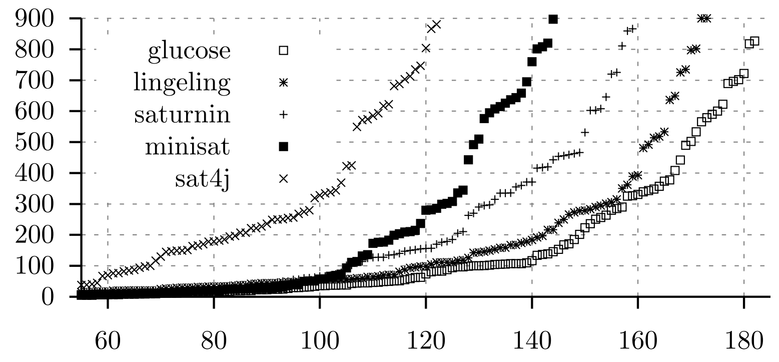
`saturnin` Developed by the author of this thesis with educational purpose, `saturnin` implements watcher structure *à-la* `MiniSat`, the average *lbd* restart policy and uses the freeze (page 41) as clause database management technique. Binary clauses uses their own occurrence list in order to keep memory tight.

Since 2002, researchers from the SAT community have organised competitions and shared results on `http://www.satcompetition.org`. Initially, the aim of those competitions was to provide a comparison of the different solving algorithms under the same environment. Progressively, those competitions evolved to provide different tracks: random generated instances, instances generated to be as hard as possible for the algorithms, and the instances provided by industrials. Later, other categories were added such as parallel solving and categories in which a file containing the unsatisfiable proof is needed. As some solvers have heuristics that perform better on some given problem, the selection of the instances is a very sensitive subject for those competitions.

In order to present their respective results, the last version was taken for each of the presented solvers (`MiniSat` 2.2, `glucose` 3.0, sat4j 2.3.5.v20130525, lingeling ayv, saturnin 09f9c00) and tested on the SAT 2011 Competition benchmark, industrial instances, using the CPU time limit of 900 seconds, as in the SAT 2011 Competition. This evaluation can be found at Figure 2.13.

The second benchmark used is the SAT 2014 Competition, industrial instances, using the time limit of 5000 seconds, as in the SAT 2014 Competition. This second evaluation can be found at Figure2.14 The hardware used for this are DELL PowerEdge 6220 using Intel Xeon E5-2643 @ 3,3GHz with at most 64 Gb of RAM.

We can see that the selection of the benchmark plays an important role in the result of the competitions. Those competitions are pictures taken at a given time, on a given set of instances and on a given hardware, and should not be taken for anything more. However, through competitions, we are able to see trends and it can also show techniques that have a real impact on the resolution of the SAT problem.

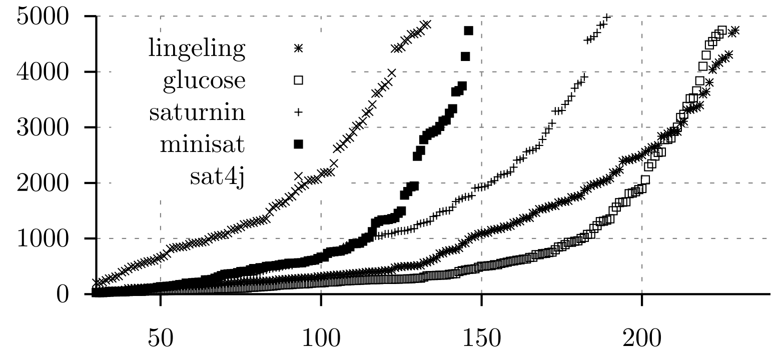


(a) Cactus plot between different state-of-the art solvers

Solver	#SAT	#UNSAT	total
glucose	**83**	**100**	**183**
lingeling	**83**	91	174
saturnin	79	82	161
MiniSat	71	75	146
sat4j	67	56	123

(b) Detailed results for some state-of-the art solvers

Figure 2.13: Evaluation of some solvers on the instances of the SAT 2011 Competition, with a CPU time limit of 900 seconds



(a) Cactus plot between different state-of-the art solvers

Solver	#SAT	#UNSAT	total
lingeling	92	**138**	**230**
glucose	**101**	125	226
saturnin	98	92	190
MiniSat	76	71	147
sat4j	79	55	134

(b) Detailed results for some state-of-the art solvers

Figure 2.14: Evaluation of some solvers on the instances of the SAT 2014 Competition, with a CPU time limit of 5000 seconds

2.4 Conclusion

In this part, different methods to solve the SAT problem. Among them, the CDCL algorithm was carefully explained. This algorithm is used in many real world application and is able to provide answers very quickly. However, the time needed to provide an answer is sometimes too long for their users. To have answers faster, different possibilities are offered to them. Among those choices, there is the possibility to use the parallel paradigm. The next chapter will provide explanation on how this paradigm can be used to solve the SAT problem.

CHAPTER 3

Parallelism

Great things are done by a series
of small things brought together.
— Vincent Van Gogh

3.1 Introduction

In 1965, at the beginnings of the semiconductor industry, Gordon Moore tried to predict the evolution of this industry over the deceny. This prediction was published [Moore, 1965] and lead to what we usually refer as Moore's Law.

This 'law' states that the number of components per integrated circuit is doubling every two years. The Figure 3.1 shows that the number of transistors on a CPU seems to follow an exponential increase, thus validating Moore's Law.

This increase of number of transistors on a CPU, combined with the increase of the clock frequency (see Figure 3.2) allowed programs to run faster and faster over the years. However, since 2005, clock frequency is almost constant. This stability of the clock frequency over the last decade can be explained by different economical factors. First, designing new chips is extreamly expensive. Second, with a higher clock frequency, more heat must be dissipated, therefore requiring more investment on the user-side of the CPU. Therefore, a lot of the extra transistors are dedicated to multitasking. This means that if we want to harvest this power, multitasking must be considered.

If we wish to consider multitasking, different theoretical notions are needed. First, we need to ask ourselves whether multitasking is fundamentally different or more powerful than the regular Turing machine. Unfortunately, parallel Turing machine does not offer more computing capabilities. They can neither solve problems that not solved by regular Turing machine, nor can they solve problems with a lower asymptotic complexity.

However, parallel Turing machine can provide an acceleration factor to a sequential system. In [Amdahl, 1967] the author has provided a base to the formulae's 3.1 and 3.2 that are known as Amdahl's law. Those provide a simple way to compute the efficiency S of an acceleration for a parallel system. Let T be the execution time of a system, T_p be the execution time of the parallel system, s the fraction of T concerned by the parallelism, and A the acceleration obtained by the parallelism.

$$T_p = (1 - s) \times T + \frac{s \times T}{A} \tag{3.1}$$

$$S = \frac{T}{T_p} = \frac{1}{(1 - s) + \frac{s}{A}} \tag{3.2}$$

3.2 Architectures

In order to implement parallel methods, we must choose what kind of parallelism we want, study the different opportunities with their respective advantages and drawbacks.

3.2.1 Multi-core

In order to increase the efficiency of their CPU, chip makers have proposed various methods among which simultaneous multi-threading. This technique uses its superscalar CPU to dispatch the different instructions from independent threads during the idle time of the processor. Its most widely known implementation is done by Intel under the name *hyperthreading*.

Another way to improve the capabilities of CPU is by providing them multiple core. On a single chip, multiple cores are embedded and there is a dedicated part of the chip to control which instruction will be performed by which process/thread. This technique allows chip makers to provide an easy and cheap way to have a better hardware support of multi-threading.

3.2.2 GPGPU

With the increasing demand from the gaming industry for 3D graphics and effects, the graphic processing unit (GPU) maker provided more and more flexibility to control the output of those chips. One of this was the introduction of the shader programs, allowing programmers to create dedicated effects for their games. Those programs take as input some geometry objects with their optional parameters (such as the colour of each vertex, some images as a texture, . . .) and modify those. The GPU makers had their system optimized already multi-threaded by using the single instruction multiple data(*simd*) paradigm. Soon, scientists and hobbyists realized that they could use the shader capabilities to perform their algorithms in parallel given that their datas could be arranged in an acceptable way for shaders. Later, GPU makers noted this trend and offered a complete and dedicated toolchain with their dedicated language extension. Of those languages, the most well known are the NVidia CUDA [Nvidia, 2007] and OpenCL [Stone et al., 2010].

3.2.3 Distributed

Finally, another way to have different instructions used at the same time is through the uses of distinct machines. Communication between them can be achieved using TCP/IP [RFC, 1981a, RFC, 1981b], OpenMPI [Gabriel et al., 2004] or any other communication protocol. Whilst being the most common communication form, distributed computing suffers from the fact that

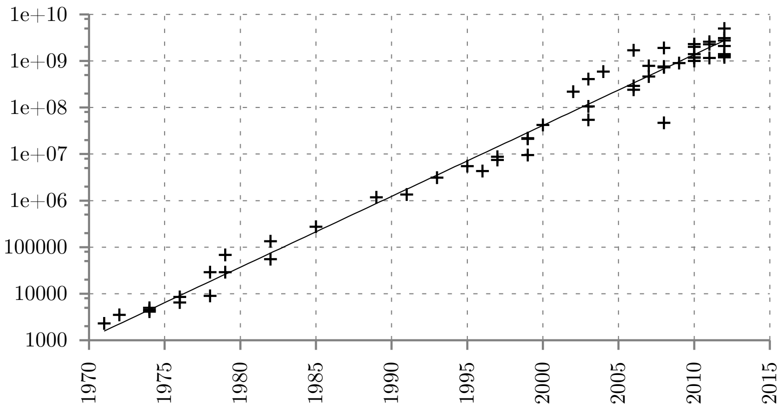


Figure 3.1: Evolution of the number of transistor on a CPU (each point represents a commercial CPU) where the abscissa is the year of introduction of the CPU and the ordinate (in logarithmic scale) represents the number of transistors. The line represent a function of type $f(x) = a^{b*x-c}$

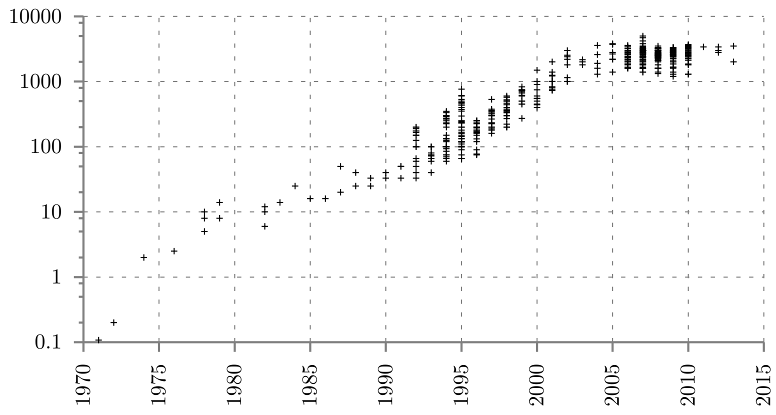


Figure 3.2: Evolution of the clock frequency (in MHz) in the CPU (each point represents a commercial CPU)

hosts are physically further apart than multi-core or GPGPU. Therefore, the communication cost is the highest in term of time between the moment when data is sent and the moment when the data is ready to be part of the computing process.

3.3 Strategies

In order to solve a problem using parallelism, different well known techniques can be applied. And for each of those techniques, they can be found outside the computing world where us, humans, try to solve a given problem.

3.3.1 Competition

The competition strategy is very well known outside the computing world, but is not very used in practice within algorithms. This method implies that the exact same task is given to different work units, each work unit having its own characteristics. By asking each of those work units to solve the problem, those are competing, as only one answer is needed. Therefore, as soon as one computing unit founds the answer, every other work unit can stop its task.

This strategy is very useful if the computing time is quite important and is very versatile. It is also possible to increase this methodology if the work units are able to communicate some intermediate results.

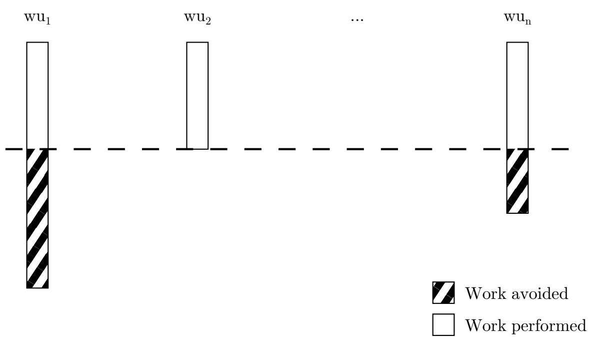


Figure 3.3: Competiting strategies between wu$_i$, $1 \leq i \leq n$, the different work units. As soon as the first work unit finds an answer, the system stops.

On a more formal description, let us assume that t is the task to be done, and $f_i(t)$ the time needed by the i-th work unit. Using the competition strategy, the time needed to solve t is $\min(f_0(t), \ldots, f_n(t))$. The acceleration would be between 1 and $\frac{\max(f_0(t),\ldots,f_n(t))}{\min(f_0(t),\ldots,f_n(t))}$.

Outside the computing world, this strategy is applied by scientific communities. When a problem needs to be solved, different groups are working on it and share their intermediate results through the publication system.

3.3.2 Pipelining

The pipelining strategy relies on the fact that we can often divide a task into sub-tasks that must be done sequentially. Each work unit is specialized for one of those sub-tasks. When a task has to be accomplished, it is given to the first work unit. Upon completion the first work unit pushes the task to the second work unit who will perfom the second sub-task, and so on. A depiction of such strategy is available in Figure 3.4.

This strategy is very useful whenever the number of tasks that need to be performed is greater than the number of work units.

Using this strategy, no acceleration can be obtained using Amdahl's law. However, if we consider the set of tasks instead of a single task, an acceleration can be observed. Let us assume that we have n tasks to perform. Each task consists in applying m distinct functions: $f_1, \ldots, f_m$, $m < n$. Let $t(f)$ the time needed for the function f. The time needed for a sequential execution is defined by Equation 3.3.

$$n \times (t(f_1) + \ldots + t(f_m)) \tag{3.3}$$

The time needed for a pipelined execution is defined by Equation 3.4. This equation can be divided in three parts:

a) initialization of the work units: first, f_1 is applied on the first task. The time needed is $t(f_1)$. Next, f_2 is applied on the first task while f_1 is applied on the second task. The time needed for this $\max(t(f_1), t(f_2))$ as the system needs that both functions have ended before going further. Next, f_3 is applied on the first task while f_2 is applied on the second task and f_1 on the third. The time needed is $\max(t(f_1), t(f_2), t(f_3))$ for the same reason as before. This will goes on until f_{m-1} is applied on the first task. Therefore, we can compute the time needed for this operation as $\sum_{i=1}^{m-1} \max(t(f_1), \ldots, t(f_i))$

b) every work unit has some work: f_1 has been performed on the $m-1$ first tasks. Therefore, f_1 has to be applied to the $n-(m-1)$ other tasks. For each of those time, the system has to wait for the slowest f_i, therefore $\max(t(f_1), \ldots, t(f_m))$.

c) treatment of the last tasks: since f_1 has been performed on every task, we must process the last tasks the same way as we treated the first ones in a)

$$\begin{gathered}\underbrace{\sum_{i=1}^{m-1} \max(t(f_1), \ldots, t(f_i)) +}_{a} \\ \underbrace{(n-(m-1)) \times \max(t(f_1), \ldots, t(f_m))}_{b} + \\ \underbrace{\sum_{i=1}^{m-1} \max(t(f_{m-i}), \ldots, t(f_m))}_{c}\end{gathered} \tag{3.4}$$

The possible acceleration that can be obtained depends on the number of tasks to treat and the difference between $t(f_1), \ldots, t(f_m)$.

This strategy is used in CPU design [Mano and Kime, 2008] in order to load data and instructions. Outside the computing world, this strategy is used in production chains.

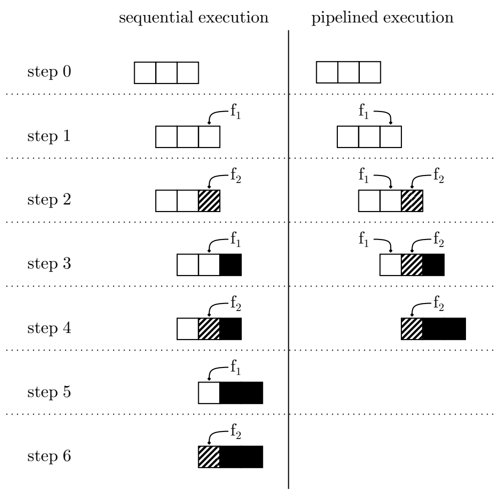


Figure 3.4: Treatment of 3 tasks where the treatment consists in applying f_1 and f_2. Both f_1 and f_2 require each 1 time unit. On the left, a sequential execution requires 6 steps. On the right, only 4 steps are required.

3.3.3 Work division

The work division strategy relies on the fact that a task can be seen as a combination of smaller tasks of the same nature. Work division requires some computing before and after the work unit have fulfilled their tasks. Before as the problem needs to be divided, and after as the results needs to be merged. Usually, those two tasks are not parallelised and from their cost depends the possible acceleration. Moreover, when the task is divided, the sub-tasks might not be solved using the same time. This is due to the different complexity of the sub-tasks or from the different computing power. Therefore, the system may have to wait for the completion of the last sub-task in order to conclude, see Figure 3.5.

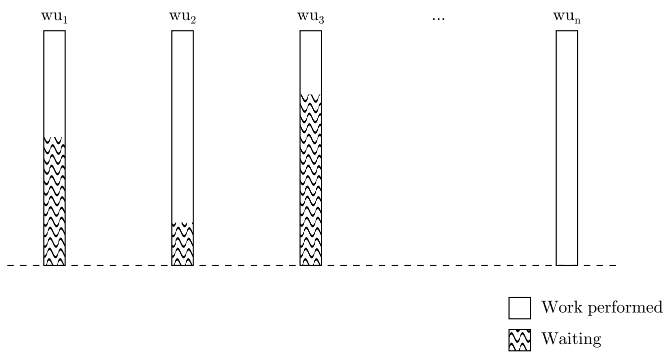


Figure 3.5: Work division strategy across different work units. The system has to wait that every work unit has finished to be able to stop

To limit the problem of uneven work division, multiple strategies can be applied. One can use load balancing schemes to divide the work of a work unit to provide some to a waiting one. Another way to limit the uneven work division is by dividing the work into n sub-tasks for the m work units, with $n >> m$. Such approach has been used in [Posypkin et al., 2012, Régin et al., 2013].

Outside the computing world, we often see this strategy of splitting the work and responsibilities. In a baseball game, the batter hits the ball and the opposite team must catch it as fast as they can. As the field is somewhat large and seven people are present to catch the ball, they are scattered to divide the field they are responsible for.

3.4 SAT solving

As many subjects in computer science, scientists are investigating the possibility to effectively use multitasking to speed up the resolution of the SAT problem. In this section, we provide information about different techniques used by the community.

3.4.1 Portfolio's

The portfolio schema is based on the competitive strategy.

The obtained results can be quite impressive as can be observed from the 2011 SAT Competition. Those can be explained by two factors: the orthogonality of the solvers and their communication scheme. Let a, b be solvers, S a set of instances, S_a the set of instances solved by solver a with limit l and S_b the set of instances solved by solver b with limit l. This limit can be the number of conflicts, the time in second, the number of operations, ... The orthogonality of a and b can be defined as:

$$orth(a, b) = \#\{x \in S | x \in S_a, x \notin S_b\} + \#\{x \in S | x \in S_b, x \notin S_a\} \tag{3.5}$$

Orthogonality of solvers have an impact on the resolution of a set of instances as "there is no free lunch" [Wolpert and Macready, 1997]. This expression states the fact that for any heuristic, an instance can always be forged to obtain the worst case scenario. Therefore, by combining different heuristics in parallel, one can hope that the number of pathological cases is reduced.

The communication scheme of a portfolio solver is the information that can be sent from one solver to another. We can divide portfolio solver into two categories. Those who concentrate their effort on the orthogonality and those who concentrate their effort on communication.

Independent solvers

Independent portfolio are dedicated to the maximisation of the orthogonality.

`ppfolio` [Roussel, 2011] which describes itself as "*equivalent to typing the following command line:* `solver1 & solver2 & solver3 & solver4 & solver5 &`", takes as input a SAT instance and feeds it underlying solver (list available shown in Figure 3.6). Even tough `ppfolio` does not provide any communication between the underlying solvers, good results were obtained at the SAT 2011 Competition[1]. One of the reasons for this is the high orthogonality of the underlying solvers, using completely different solving techniques through some state of the art solvers.

[1]`http://www.satcompetition.org/`

- `cryptominisat` [Soos, 2010]
- `lingeling` [Biere, 2013]
- `clasp` [Gebser et al., 2007]
- `TNM` [Wei and Li, 2009]
- `march_hi` [Heule and van Maaren, 2009a].

Figure 3.6: List of solvers included in `ppfolio`

Communicating solvers

When using multithreading, it is possible for the threads/solvers to easily communicate to each other. Such message can be the clauses generated after each conflict. Current communicating portfolio have less orthogonality as they use the same core solver, replicated with variance in their heuristics or by changing constant values in them.

Let n be the number of threads, each thread will send its clause to every other thread. As a result, after each thread has reached its first conflict, the number of clauses/messages sent is $\mathcal{O}(n)$. Different studies have been done in order to reduce this effect with a high number of workers.

ManySat In [Hamadi et al., 2009a], the authors of `ManySat` studied the rate at which clauses are exchanged. They discovered that the number of clauses that fit their initial transfer condition (clauses with less than 8 literals) diminish over time. If the number of clauses get lower, the communication also diminishes with a static transfer condition. Therefore, they adapted the TCP congestion avoidance algorithm to have a dynamic length condition for the transfer. Whenever the rate at which clauses of size s are exported becomes too low (high), s is incremented (decremented).

plingeling In [Biere, 2013], the author of `Plingeling` has evolved his communicating scheme from unit clauses to clauses with a *lbd* value lower than 8 and a size lower than 40.

syrup In [Audemard and Simon, 2014], the authors uses a slightly more complex and dynamic exchange rule. Instead of taking arbitrary constant for the *lbd* value for exported clauses, the maximum value is the median one. As for the size criterion, clauses must have a size lower than the average size to be exported. Finally, their last export criterion is that in order to be exported, a clause must have been used in the conflict analysis procedure at least twice.
For imported clauses, those are set in special occurrence lists using only 1 watcher (see page 27). This avoids missing an unsatisfied clause but still limits the impact of those imported clauses. Once a clause becomes unsatisfied, it is incorporated in the two watched scheme.

SArTagnan In [Kottler and Kaufmann, 2011], the authors propose a portfolio solver using distinct solving mechanism per thread, and each of this thread may communicate learnt clauses to each other. Those techniques are CDCL solver with changes in heuristics, *Decision Making* *with Reference Points* [Goldberg, 2006, Goldberg, 2008], clause subsumption, blocked clause elimination [Järvisalo et al., 2010] and variable elimination [Eén and Biere, 2005]. Variable elimination is implemented in only one thread in order to avoid problems such as having the same variable removed twice.

Bandit ensemble One of such work is [Jabbour et al., 2012], it uses *bandit ensemble for parallel SAT solving*, BESS. BESS attaches a *multi-armed bandit* (MAB) to each thread t, where each arm represents an emitting thread to t. In each time period, the individual MAB computes the reward for each emitting thread. From this reward, an *aliveness threshold* is computed to check whether the considered emitting thread should be turned into a sleeping thread. Such sleeping thread is a thread that will not send clause to the thread t. For each *alive* thread set to sleep, the system will *awake* the one that has been sleeping for the longest time.

Community branching Based on [Audemard et al., 2012b], the work presented in Chapter 4, the authors of [Sonobe et al., 2014] try to increase the orthogonality of the threads. To do this, they first try to identify community structures in an instance [Ansótegui et al., 2012]. A community is a set of variables which share a high number of clauses between the different variables of the community, and almost none with variables outside the community. Once communities are found, each thread is assigned one particular community to increase orthogonality based on search space. In practice, the value for the variable selection heuristic (see Section 2.2.1) is increased before each restart. Using this method, variables of the selected communities will be selected first.

3.4.2 Divide and conquer

Divide an conquer solvers are based on the work division strategy. Different approaches have already been used. Some of them are exposed in the following section.

Guiding path

They are all based on the concept of search space division. Those subspaces are assigned to different SAT worker as a form of *guiding path* [Zhang et al., 1996].

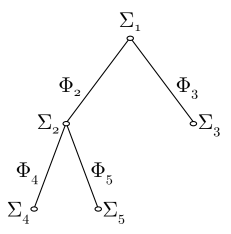


Figure 3.7: Guiding tree for the instance Σ

A guiding path is a formula added to the instance Σ in order to divide the search space. This guiding path is created by recursion. First, we define Σ_1 as the undivided instance we want to solve Σ, and therefore $\Sigma_1 = \Sigma$. For the ease of the explanation we want to define Σ_i as a conjunction of some formulae ϕ and Σ. Thus, we define Σ_1 as $\Sigma = \Sigma \wedge \Phi_1$ and therefore $\Phi_1 \equiv \top$. Second, we divide Σ_1 in two sub-spaces to obtain Σ_2 and Σ_3 by using respectively the formula Φ_2 and Φ_3. Later, when a Σ_i needs to be divided into $\Sigma_{i\times 2}$ and $\Sigma_{i\times 2+1}$, its respective division formula, or guiding path, will be $\Phi_{i\times 2} \wedge \Phi_i \wedge \Phi_{i/2} \wedge \ldots \wedge \Phi_1$ and $\Phi_{i\times 2+1} \wedge \Phi_i \wedge \Phi_{i/2} \wedge \ldots \wedge \Phi_1$. The conjunction of those guiding path can be represented as a tree, called the guiding tree. An example of such depiction is shown at Figure 3.7, applied on the instance Σ. From the figure, we can deduce $\Sigma_2 = \Phi_2 \wedge \Sigma$, $\Sigma_3 = \Phi_3 \wedge \Sigma$, $\Sigma_4 = \Phi_4 \wedge \Phi_2 \wedge \Sigma$, $\Sigma_5 = \Phi_5 \wedge \Phi_2 \wedge \Sigma$.

sat@home

The `sat@home` [Posypkin et al., 2012] team research aims at solving instances representing a cryptographic problem. Their operating mode is as follow: first they choose 32 variables $v_0, \ldots, v_{31}$ from the initial instance Σ. From those, they create 2^{32} instances of the form $\vec{v} \wedge \Sigma$ where $\vec{v}$ is one of the 2^{32} possible assignation vectors for the variables $v_0, \ldots, v_{31}$. Finally, those instances are solved using a SAT solver on a cluster of computers available through *volunteer computing*. In this process, the selection of the variable is an extremely important phase, as the work balance between the different computing units depends on it. The sat@home team proposed in [Semenov and Zaikin, 2013] an algorithm based on a Monte-Carlo approach to select them. Their aim was to minimize the running time of each of the 2^{32} jobs. To do this, they generate a set of variables and for this set, they generate a small subset of possible assignations. Those subsets are solved and their running time kept in order to compute a possible average running time for all possible assignations. When those average running times are known, they keep the set of variables that seems the most promising.

Part-Tree Learning

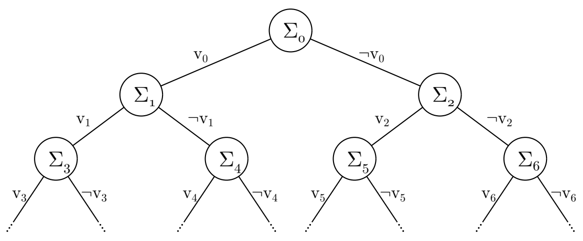


Figure 3.8: A partitioning tree

Another implementation of the work division strategy has been done in the solver `ptl` by the authors of [Hyvärinen et al., 2011]. `ptl` creates a partition tree as depicted in Figure 3.8. In that figure, Σ_i is a partition of the instance and the partition is done through the instantiation of the variables between Σ_0 and Σ_i. Initially, the n work units solve the n first partitions: $\Sigma_0, \ldots, \Sigma_{n-1}$. When a work unit finishes its task, it provides the answer to a master. The master processes this answer as it might lead to stopping other work units. Indeed, if both $\Sigma_{2\times i}$ and $\Sigma_{(2\times i)+1}$ are solved, Σ_i has been solved and can be stopped. On the other hand, when Σ_i is solved, any of its sub-partitions can be considered as solved. Finally, the idling work unit solves the next unsolved partition that has not been assigned yet.

During the search, the work units of `ptl` generate different clauses as it is based on a CDCL solver. As `ptl` uses the assumption mechanism (see 2.2.7), those can be kept in memory whenever a work unit changes the partition it is working on.

pcasso

In the solver `pcasso`, the authors of [Irfan et al., 2013] take as base solver `ptl` and have modified the partition function to use the scattering rule proposed in [Hyvärinen et al., 2006] depicted in Equation 3.6 in which v_i are variables.

$$\Sigma_i = \begin{cases} \Sigma, & \text{if } i = 0 \\ \Sigma \wedge v_1, & \text{if } i = 1 \\ \Sigma \wedge \neg v_1 \wedge \ldots \wedge v_{i-1} \wedge v_i, & \text{if } 1 < i < n \\ \Sigma \wedge \neg v_1 \wedge \ldots \wedge v_{i-1} \wedge \neg v_{n-1}, & \text{if } 1 = n \end{cases} \tag{3.6}$$

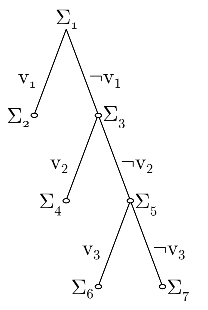


Figure 3.9: The partition tree used by `pcasso` for 7 work units

As for every divide and conquer strategy, a key element is their dividing scheme. In `pcasso`, the different T_i used in Equation 3.6 are variables selected using look-ahead technique. The selected metric is the *mixdiff* [Heule and van Maaren, 2009b] which is based on the number of propagated literals and the number of binary clauses created.

cube and conquer

The cube and conquer method [Heule et al., 2012] uses cubes (Section 1.1.2) as partition methodology. The main idea is that in order to solve hard industrial instances, they partition the search space using cubes. In order to create those cubes, a look-ahead procedure is used at the start of the program. One of the key element is that this look-ahead procedure requires dedicated heuristics: one for the cut-off, that is the length of the cube, and the second is the heuristics around the splitting, that is the variable and the value selection.

As cut-off heuristics, the authors of [Heule et al., 2012] use as threshold a combination of the depth of the current branch and the number of implied literals. This combination has to be higher than an evolving percentage of the number of variables. As splitting heuristics, they use the number of variables propagated by assigning a truth value to a variable.

When the different cubes are created they can be used with incremental SAT techniques (assumptions) as solving the conjunction of the instance and a cube can be seen as one job. The authors have asked themselves how to solve those jobs in a parallel environment. One possibility is to use a portfolio strategy on each job. Using this idea, if we dispose of n computing units then each works on the same job, and the next job is considered only when the current job is solved. Another possibility is to give to each of the computing unit a different job to solve. It is the latter one that was implemented in the solver `iLingeling`.

This idea was later improved in [van der Tak et al., 2012]. Their motivation was to improve the heuristic to avoid the generation of too many or too few cubes. This improvement is done by running a CDCL solver along the cube phase to determine whether the current cube can be easily solved or not.

3.4.3 Pipelined

MTSS

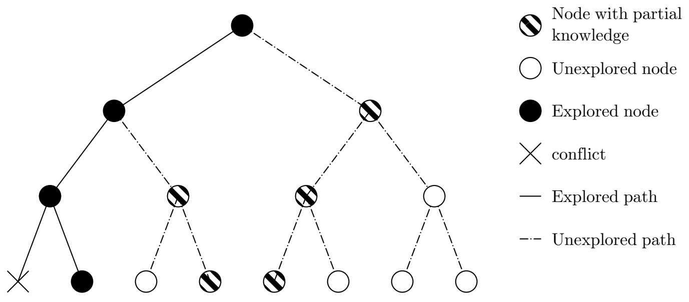


Figure 3.10: The search tree of MTSS

In [Dequen et al., 2009,Vander-Swalmen et al., 2011], the authors present algorithms capable of solving any SAT formula by parallelizing a DLL search on multicore architectures called Multi-Threaded SAT Solver (MTSS). In a DLL search multiple nodes are explored and for each node, a variable is assigned. The selection of this variable can be quite time consuming. Therefore, the authors define the following framework to make use of the parallelism. A rich thread explores the search tree. At each node, a look-ahead technique is used to select the next variable to branch on. Next to this rich thread exists poor threads whose job is to pre-process the different parts of the search tree. These pre-processes can have different forms and produce different information such as the state of the given node, the variable to branch on, the value (SAT/UNSAT) of the node, . . . When the rich thread wants to explore a new node, three different situations can appear. First, no partial knowledge is present. In such a case, the rich continues its search. Second, partial knowledge is present. The knowledge is processed. Finally, a poor thread is busy on that particular node. In such case, the poor thread is promoted to a rich thread and the rich thread is demoted to a poor thread and starts processing an unexplored node. This makes a parallel solver based on the pipelined strategy (poor threads make some pretreatment for the rich thread) that aims at solving an instance using the divide and conquer paradigm.

Using GPGPU

As presented in section 3.2.2, modern computers have access to a co-processor, the general purpose graphical processing unit (GPGPU). This co-processor can be used in a pipelined strategy where it would compute a value for some heuristic during the search of the main CPU.

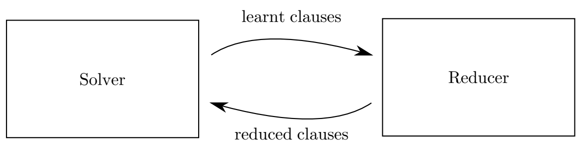


Figure 3.11: The solver/reducer architecture

In [Gulati and Khatri, 2010], the authors propose a methodology based on survey propagation as variable heuristic. Each time a given percentage of the variables is assigned, a call to a survey propagation algorithm implemented on the GPGPU is made. The result of this call is a set of literals that can be used for the branching as they should have a high probability to be assigned to $\top$.

Clause strengthening

Pipelining is also used in [Wieringa and Heljanko, 2013]. In this paper, the authors present the following schema. The system consists of two key elements: the solver and the reducer, as depicted in Figure 3.11. The solver fills a heap with its learnt clauses. Those are sorted according their *lbd* value. In order to limit the size of that heap, a maximum size is set. When the solver tries to add a clause to a filled up heap, the oldest clause is removed to free up some place. When the reducer requests a clause, the heap provides the one with the best *lbd* value. This choice provide the reducer both “fresh” and “good” clauses.

As for the reducer, different techniques can be applied such as vivification [Piette et al., 2008]. Once the clauses are reduced, they can be added back to the solver. To incorporate the reduced clauses, the solver can check whether a backtrack is needed (the clause is unsatisfied by the current partial interpretation) or if it just has to be added to the occurrence list.

3.5 Conclusion

As we have seen, there are many different ways to implement parallel/distributed search for SAT using the CDCL algorithm. However, many implementations either relies on the physical architecture or divide their work using literals or conjunctions of it. This has as side effect to prohibit using clauses to divide the work.

CHAPTER 4

Penelope: a portfolio based solver

A system is more than the sum of its parts; it is an indivisible whole. It loses its essential properties when it is taken apart. The elements of a system may themselves be systems, and every system may be part of a larger system.

— Russel L. Ackoff

This chapter presents the work that lead to the publications [Audemard et al., 2012b, Audemard et al., 2012c] and some technical reports [Audemard et al., 2012a, Audemard et al., 2013a, Audemard et al., 2014a]

4.1 Introduction

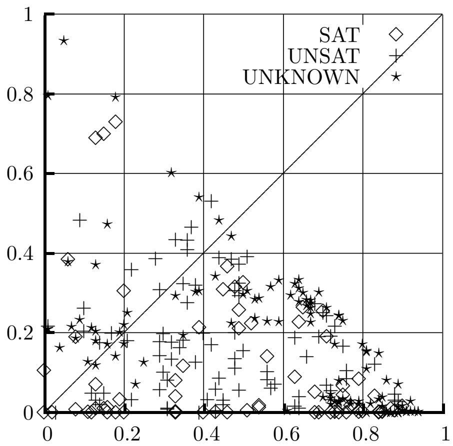
The problem of predicting the usefulness of a given learnt clause is known in the sequential SAT solving. Some even claim that "90% of the time spent by a CDCL is useless" [Simon, 2014]. As the time needed by propagation depends on the number of learnt clauses, having an effective learnt clauses management strategy is crucial to performances. In a portfolio, each thread generates new clauses and shares them with the other threads. This increase even more the need for an effective clause management strategy if we want a scalable solver.

The *psm* measure ([Audemard et al., 2011a] and page 41) has been proposed to dynamically manage learnt clauses. Roughly, it consists in comparing the current (partial) interpretation to the set of literals of each learnt clause. The main idea is the following: if the set-theoretical intersection of the current interpretation and the clause is large, then the clause is unlikely useful in the current part of the search space. On the contrary, if this intersection is small, then the clause has a lot of chance to be useful for unit-propagation, reducing the search space. This measure has been used in a strategy to manage the learnt clauses database which enables to *freeze* a clause, namely to remove it from the set of learnt clauses on a temporary basis, when it is considered "useless". Periodically all clauses are re-evaluated in order to be frozen or activated. This technique proves very efficient in an empirical point of view, and succeeds to select relevant learnt clauses.

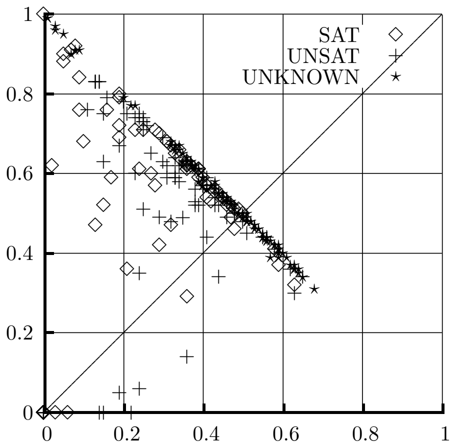
4.2 A premiliminary experimentation

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

Accordingly, we wanted to know how useful are the clauses shared in a portfolio-based solver. To this end, we ran some experiments on a dual socket `Intel XEON X5550` quad-core 2.66 GHz with 8 MB of cache and a RAM limit of 32GB, under Linux CentOS 6 (kernel 2.6.32). All solvers use 8 threads. The timeout was set to 1200 seconds WC for each instance. If no answer was delivered within this amount of time, the instance was considered unsolved. We used the application instances (300) of the SAT competition 2011. Those experiments were using a state-of-the-art portfolio-based SAT solver. We choose the solver `ManySat` 2.0 (based on `Minisat 2.2`), because in this solver, the only difference between the working threads are caused by the first decision variables which are selected randomly. Except this initial interpretation, each CDCL worker exhibits the exact same behaviour (in terms of restart strategy, branching variable heuristics, etc.), which allows us to make a fair comparison about clause exchange without any side effect. Hence, it represents a good framework to deal with parallel SAT solvers and clauses sharing. By default all clauses of size less than 8 are shared. Moreover, `ManySat` provides a deterministic mode [Hamadi et al., 2011]. Let us emphasize that we have activated this option to make the obtained results fully reproducible and we report the detailed results online[1].



(a) `ManySat`



(b) aggressive `ManySat`

Figure 4.1: Comparison between useful share clauses and unused deleted clauses. Each dot corresponds to an instance. x-axis gives the rate of useful shared clauses $\#used(\mathcal{SC})/\#\mathcal{SC}$, whereas the y-axis gives the rate of unused deleted shared clauses $\#unused(\mathcal{SC})/\#\mathcal{SC}$.

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

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

The Figure 4.1 synthetizes the results obtained during this first experiment. The results are reported in the following way: each point of Figure 4.1 is associated with an instance, and the x-axys corresponds to the rate $\#used(\mathcal{SC})/\#\mathcal{SC}$, whereas the y-axys corresponds to the rate $\#unused(\mathcal{SC})/\#\mathcal{SC}$, and we report the average rate over the different threads. Figure 4.1(a) gives the results for `ManySat`. First of all, we can remark that the rate of useful shared clauses differs greatly over different instances. We can also note, that in a lot of cases, `ManySat` keeps shared clauses during the entire search (dots near the x-axys). This is due to the non-aggressive cleaning strategy of `MiniSat` where in many instances no cleaning is performed. Threads can keep useless clauses a long time and have to support an over cost without any benefit.

We have conducted the same experiment with a much more agressive cleaning strategy. We have choosed the one presented in [Audemard et al., 2011a] (see page 41) and we report the result in Figure 4.1(b). Here, in many cases, shared clauses are deleted without any usage and the percentage of shared clauses that are used at least one time decreases with respect to the basic version of `ManySat`. These results can be explained quite easily. If only few cleanings are done, the threads have to manage a lot of useful *and* useless shared clauses. In one hand, it owns a lot of information about the problem to solve, and propagates many units clauses. On the other hand, such a solver has to maintain a large number of clauses uselessly, which greatly slows down its exploration.

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

4.3 Selection, sharing and activation of good clauses

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

- Imported clauses can be subsumed by clauses already present in the database. Since subsumption computation is time consuming, it is necessary to give the possibility to remove periodically learnt clauses.
- Imported clauses may be useless during a long time, and suddenly become useful.

- Each thread has to manage many more clauses.
- Characterizing good imported clauses is a real challenge.

For all of these reasons, we propose to use the dynamic management policy of learnt clauses proposed in [Audemard et al., 2011a] inside each thread. This recent technique enables to activate or freeze some learnt clauses, imported or locally generated. The advantage is twofold. The overhead caused by imported clauses is greatly reduced since clauses can be frozen. Nevertheless, clauses estimated useful in the next future of the search are activated. Let us present more precisely this method in the next Section.

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

We present in the next Section a technique where both the sender and the receiver of a clause have a strategy. Obviously, any sender (export strategy) tries to find in its own learnt clause database the most relevant information to help the other workers. However, the receiver (import strategy) here does not accept the shared clauses in a blind way. We have called our case study solver `PeneLoPe`[2] (**P**arallel **L**bd **P**sm solver).

Importing clause policy

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

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

Exporting clause policy

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

- *unlimited*: any generated clause is exported towards the different threads.
- *size limit*: only clauses whose size is less than a given value are exported [Hamadi et al., 2009a] (8 in the article and our experiments) .

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

- *lbd limit*: a given clause c is exported to other threads if its *lbd* value $lbd(c)$ is less than a given limit value d. In our experiments, we used the same value as the one in the *size limit* strategies. Let us also note that the *lbd* value can vary over time, since it is computed with respect to the current interpretation. Therefore, as soon as $lbd(c)$ is less than d, the clause is exported.

Restarts policy

Beside exchange policies, we define two restart strategies.

- *Luby*: Let l_i be the i^{th} term of the Luby serie [Luby et al., 1993b]. The i^{th} restart is achieved after $l_i \times \alpha$ conflicts (α is set to 100 by default).

- *LBD* [Audemard and Simon, 2009a]: Let LBD_g be the average value of the LBD of each learnt clause since the beginning. Let LBD_{100} be the same value computed only for the last 100 generated learnt clause. With this policy, a restart is achieved as soon as $LBD_{100} \times \alpha > LBD_g$ (α is set to 0.7 by default). In addition, the VSIDS score of variables that are unit-propagated thank for a learnt clause whose *lbd* is equal to 2 are increased, as detailled in [Audemard and Simon, 2009a].

We have conducted experiments to compare these different import, export and restart strategies. We ran these different versions and Table 4.1 presents a sample of the obtained results This table reports for each strategy the number of SAT instances solved (#SAT), together with the number of UNSAT instances solved (#UNSAT) and total (#SAT + #UNSAT).

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

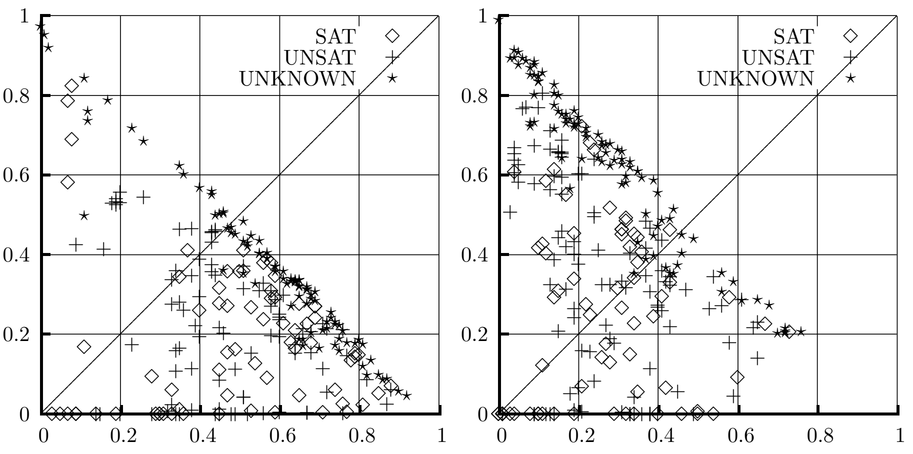
This was the reason why a size-based limit was introduced with the idea that the smallest clauses produce the best syntactic filtering, and therefore are preferable. Indeed, in Table 4.1, it appears clearly that "size limit" (clauses containing less than 8 literals) policy outperforms the "unlimited" one. This simple limit shows its usefulness, but a main drawback is that it has been shown [Beame et al., 2004] that longer clauses may greatly reduce the size of the proof.

Using the *lbd* value $lbd(c)$ of a clause c can improve the situation as $lbd(c) \leq \text{size}(c)$. Hence, if the same value v is used for both the size and the *lbd* limits, more clauses are exported with the *lbd* policy. So, specifying a limit on the *lbd* allows us to import larger clauses if those ones are heuristically considered as promising. This could represent a problem for a parallel solver without the ability to freeze some clauses. Nevertheless, as `PeneLoPe` contains such mechanism, the impact is greatly reduced. From an empirical point of view, Table 4.1 shows that the "lbd limit" obtains the best results among all exporting strategies. We have also tried to limit the export to unary clauses (line *size=1*) like most current portfolio solvers do, but this does not lead to good performance, since only 177 instances are solved.

Let us now focus on the restart strategy. Even tough the *luby* technique performs better on SAT instances, it obtains overall worst results than the "*lbd* " one. This clearly shows the

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

Table 4.1: Comparison between import, export & restart strategies using deterministic mode



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

Figure 4.2: Comparison between usefull share clauses and useless deleted clauses. Each dot corresponds to an instances. x-axis gives the rate of useful shared clauses $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, whereas the y-axis gives the rate of unused deleted shared clauses $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

particular interest of this *lbd* concept introduced in the solver described in [Audemard and Simon, 2009a]. About import strategy, no clear winner appears when looking at the results in Table 4.1. Indeed, the best results in term of number of solved SAT instances is obtained with *no freeze* (97) when associated with the "luby" restart and the "size limit" export strategy, whereas the best number of solved UNSAT instances is obtained with the *"freeze"* strategy (113). Furthermore, *no freeze* enables to obtain the best overall result solving 205 instances out of the 300 used ones. Hence, it would be audacious to plead for one of the 3 suggested techniques. However, a large number of our suggested policies performs in practice better than "classical" clause exchange techniques, represented in Table 4.1 by `ManySat`.

In a second experiment, we wanted to assess the behavior of the solver when using some of our suggested policies. To this end, we have conducted the exact same experiments than the ones presented in Section 4.2; the obtained results are reported in the Figure 4.2. First, we have tried with *size limit*, *luby*, and *no freeze* policies (denoted `SLN`, see Figure 4.2(a)). Clearly, this version behaves very well, since most of the dots are located under the diagonal. Moreover, for most instances, $\frac{\#used(\mathcal{SC})+\#unused(\mathcal{SC})}{\#\mathcal{SC}}$ is close to 1 (dots near the second diagonal), which indicates that the solver does not carry useless clauses without deleting them. Most of them proves useful, and the other ones are deleted.

Then, the experimentation was conducted with the *lbd* limit, *lbd* and *freeze* combo (denoted `LLF`, see Figure 4.2(b)). At first sight, the behavior is here less satisfying than the `SLN` version, since for most instances at least half of imported clauses are deleted without being of any help. Actually, in this version, a much larger number of clauses are exported due to the "*lbd* limit" export policy, which leads to a lower rate of useful clauses. Fine-tuning parameters (*lbd* limit values, number of time a clause has to be frozen before being permanently deleted, etc.) might improve this behavior as observed on solvers of the Configurable SAT Solver Challenge (CSSC) 2014.[3]

[3]http://aclib.net/cssc2014/index.html

Looking at some detailled statistics provided in Table 4.2, it indeed appears that the `LLF` version shares a lot more clauses than the `SLN` one (column nb_u). Note that this Table contains some other very interesting information. For instance, it allows to see that for some benchmarks (e.g. `AProVE07-21`), about 90% of imported clauses are actually frozen and do not immediately participate to the search, whereas for other instances, we face the opposite situation (`hwmcc10...`) with only 10% of clauses that are frozen when imported. This reveals the high adaptability of the *psm* measure. Let us focus now on the number of imported clauses, compared to the number of conflicts needed to solve the instance. The `SLN` version very often produces more conflict clauses than it imports from other working threads ($nb_c/nb_i < 1$), even though this is not true with some benchmarks (e.g. `AProVE07-21`, `hwmcc10...`). Note that the nb_c/nb_i rate of the `LLF` version exhibits a very high variability, from 0.58 for the smallest value in Table 4.2 (`velev-pipe-o-uns...`) to more than 4, meaning that in such cases, each time the solver produces a conflict (and consequently a clause), it imports more than 4 clauses on average. Let us also emphasize that the computational cost of the *psm* measure is not major (see "*psm* time" column). During all our experiments, `PeneLoPe` have spent at most 5% of the solving time to compute *psm*.

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

4.4 Comparison with state of the art solvers

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

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

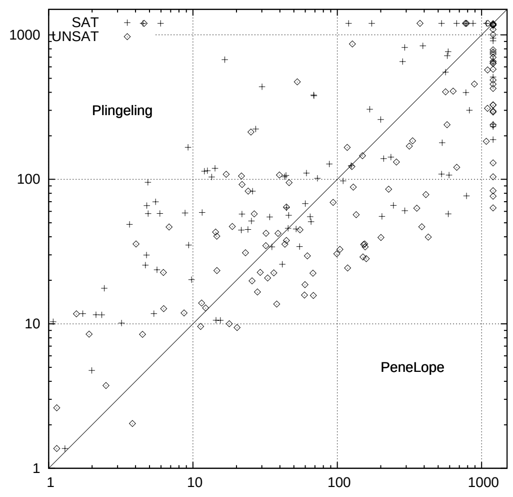
Figure 4.3 shows the obtained results through different representations; Table 4.3(a) provides the number of solved instances for the different solvers, Figure 4.3(b) details the comparison of `PeneLoPe` and `Plingeling` through a scatter plot, and a cactus plot in Figure 4.3(c) gives the number of solved instances w.r.t. the time (in seconds) needed to solve them. `PeneLoPe` outperforms all other parallel solvers; indeed, it succeeds in solving 216 instances while no other solver is able to exceed 200 (Table 4.3(a)). Note that when only considering SAT instances, the best results come from `plingeling` which solves 99 instances. This is particularly noticeable in Figure 4.3(b) where `PeneLoPe` and `plingeling` are more precisely compared; indeed, most of "SAT dots" are located above the diagonal, illustrating the strength of `plingeling` on these instances. However, results for SAT instances are closer from each other (97 for `PeneLoPe`

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

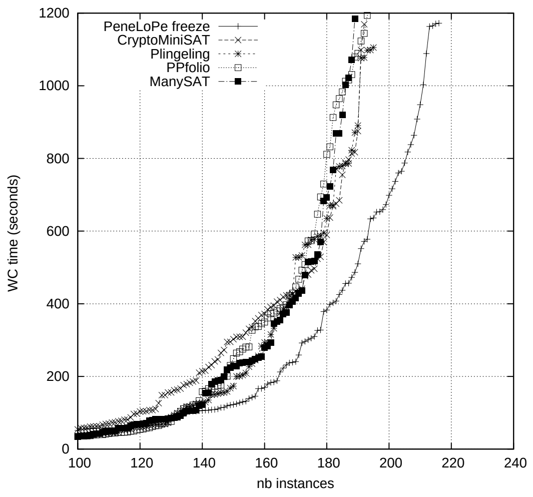
Table 4.2: Statistics about some unsatisfiable instance solving. For each instance and each version, we report the WC time needed to solve it, the number of conficts (nb_c, in thousands), the number of imported clauses (nb_i, in thousands) with between brackets the rate between nb_i and nb_c, the percentage of clauses frozen at the import (nb_f), the percentage of useful imported clauses (nb_u) and the percentage of unused deleted clauses (nb_d). Finally, we provide the rate of time (w.r.t. the overall solving time) spent on computing the psm value. Except for time, we compute the average between the 8 threads for these statistics.

Solver	#SAT	#UNSAT	total
`PeneLoPe` *freeze*	97	**119**	**216**
`PeneLoPe` *no freeze*	96	**119**	215
`plingeling` [Biere, 2013]	**99**	97	196
`ppfolio` [Roussel, 2011]	91	103	194
`cryptominisat` [Soos, 2010]	89	104	193
`ManySat` [Hamadi et al., 2009b]	95	92	187

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



(b) `PeneLoPe` *freeze* VS `Plingeling`



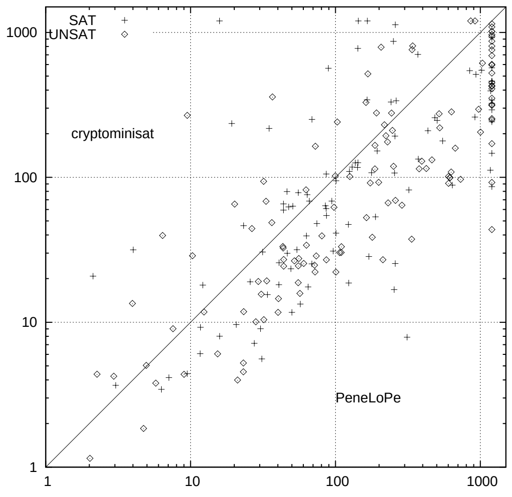
(c) Cactus plot

Figure 4.3: Comparison on 8 cores

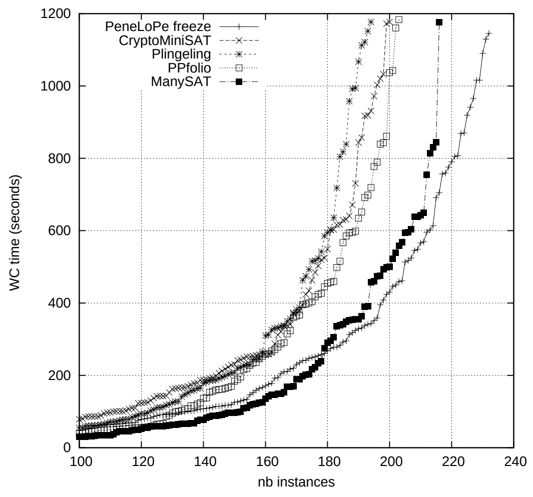
freeze, 95 for `ManySat`, etc.), the gap being more important for UNSAT problems.

Solver	#SAT	#UNSAT	total
`PeneLoPe` *freeze*	104	127	**231**
`PeneLoPe` *no freeze*	99	**131**	230
`ManySat` [Hamadi et al., 2009b]	105	111	216
`ppfolio` [Roussel, 2011]	**107**	97	204
`cryptominisat` [Soos, 2010]	96	105	201
`Plingeling` [Biere, 2013]	100	95	195

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



(b) `PeneLoPe` *freeze* VS `cryptominisat`



(c) Cactus plot

Figure 4.4: Comparison on 32 cores

In addition, we have compared the same solvers on a 32 cores architecture. More precisely, the considered hardware configuration is now `Intel Xeon CPU X7550` (4 processors, 32 cores) 2.00GHz with 18 MB of cache and a RAM limit of 256GB. The software framework is the same as with previous experiments. Each solver is run using 32 threads, and the obtained results are displayed in Figure 4.4 in a similar way than previously. First, let us remark that except for `plingeling`, all solvers improve their results when they are run with a larger number of threads. The benefit is limited for certain solvers, however. For example, `cryptominisat` solves 193 instances with 8 threads, and 201 instances with 32 threads. The improvement is stronger with `PeneLoPe` whose both versions solve 15 extra instances when 32 threads are used, and especially for `ManySat` with a gain of 29 instances. The gap can be more remarkable looking at the cactus plot in Figure 4.4(c), since our 3 competitors solve about the same number of instances within the same time (curves very close to each other), whereas the curves of `PeneLoPe` and `ManySat` clearly show their ability to solve a larger number of instances within a more restricted time. Besides, it is worth noting that `PeneLoPe` solves the same number of instances as `Plingeling`, `ppfolio` and `cryptominisat` with a (virtual) time limit of only 400 seconds. Finally, we can also notice than `PeneLoPe` can be improved on SAT instances. Indeed, it appears that Luby restarts are more efficient for SAT than for UNSAT, whereas the exact opposite phenomenon happens for UNSAT instances with the *lbd* restart strategy.

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

Finally, let us emphasize that during all our experiments with `PeneLoPe`, all working threads share the exact same parameters and strategies, just like in our preliminary experimentation in Section 4.2. Improving diversification in the different sequential CDCL searches should probably boost even more our case study solver.

4.5 Scalability

Now, let us describe the scalability of the proposed portfolio. With a portfolio that does not communicate, the acceleration might be null. Let us suppose that we have the following set of configurations $\mathcal{C}_1, \ldots, \mathcal{C}_n$ and given i threads, the configurations $\mathcal{C}_1, \ldots, \mathcal{C}_i$ are selected. If the portfolio does not communicate and the configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ are the best in 99% of the cases, adding more configurations might be beneficial for only 1% of the instances. This means that for the vast majority of the cases, adding more resources will not provide any speed-up. On the other hand, if the portfolio allows communication of the learnt clauses between the different threads, a speed-up can still be observed. This can be seen in figure 4.5. This figure was obtained by running two versions of `PeneLoPe`, using the same default provided configuration file. Therefore, no new configuration is added by adding more cores. The set of instances is the instances used in the SAT competition 2011 and the hardware used are Intel XEON X7550 with 4 octo-core @ 2GHz, 32 Gb of RAM. To have a point of comparison with a sequential solver, we include the result of the solver `glucose` (see page 44).

We can clearly see that most of the instances are solved faster using 32 threads compared to 8 threads. Moreover, the gain from communication is clearly seen on the UNSAT instances as most of the gains in terms of instances comes from the UNSAT ones. The reason for this is that we can consider that with communication, we are able to produce much more clauses per seconds and conceptually allowing to close by resolution the set of clauses faster.

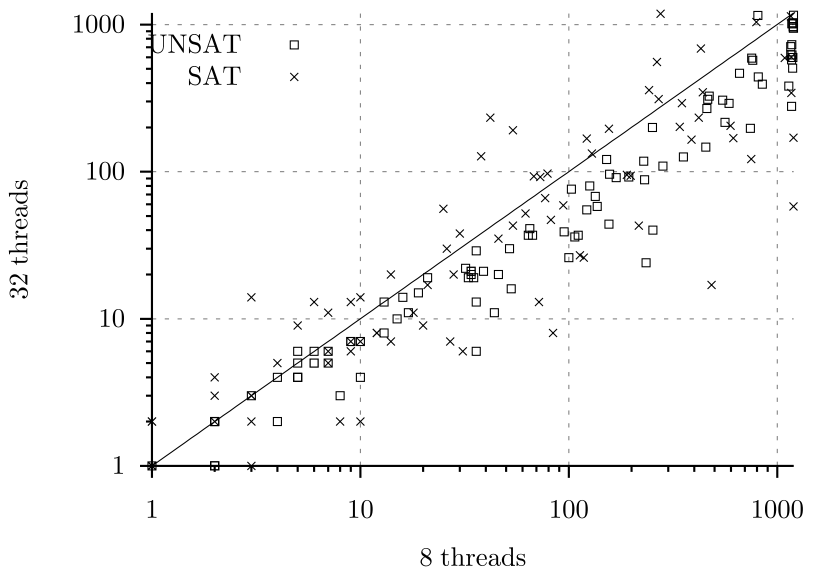
4.6 Diversification

In order to increase the orthogonality of the different threads, different parameters can be tuned for each thread. This configuration can be done through a configuration file which allows the following parameters.

exportPolicy the choice of export policy. Possible values *lbd*, *unlimited*, *legacy*

maxLBDExchange if we export according to the lbd value, the maximum value allowed to be exported

importPolicy the choice of import policy. Possible values *freeze*, *no-freeze*, *freeze-all*



(a) Scatter plot comparing wallclock time

Solver	SAT	UNSAT	total
`PeneLoPe` 32	**102**	**121**	**223**
`PeneLoPe` 8	101	109	210
`glucose`	83	100	183

(b) Detailed results

Figure 4.5: Comparison of penelope using 8 and 32 cores on the benchmark from the SAT 2011 competition

rejectAtImport allow to reject clauses at import according to their lbd value. Possible values *true/false*

rejectLBD if clauses can be rejected according to their lbd, the maximum lbd value allowed

initPhasePolicy the initial value for the phase of variables (see). Possible values: *true*, *false* and *random*

usePsm allow the use of the freezing of clauses (see page 41). Possible values: *true/false*

maxFreeze the maximum number of times a clause can be frozen if psm is used

initialNbConflictBeforeReduce the initial value of the frequency of psm clause database reduction (t_0 in Equation 2.6, see page 42)

nbConflictBeforeReduceIncrement the increment value of the frequency of psm clause database reduction (the inc value in Equation 2.6)

maxLBD the maximum lbd value allowed if psm is used for the clause database manangement

restartPolicy select the restart strategy between *avgLBD* (see page 40), *luby* (see page 40), *picosat* (see 40)

restartFactor when using average lbd restart, the value of K in Equation 2.3 (see page 40)

historicLength when using average lbd restart, the number of lenght of the short average ($\overline{L_x}$ in Equation 2.3).

lubyFactor the factor used to multiply the values of the luby suite in the case of the luby restart policy

picobase the initial base for the picosat restart policy

picolimit the initial limit for the picosat restart policy

picolimitFactor the initial limit update factor for the picosat restart policy

lexicographicalFirstPropagation propagate using the lexicographical order of the variables or a random variable for the first variable choice. Possible values *true/false*

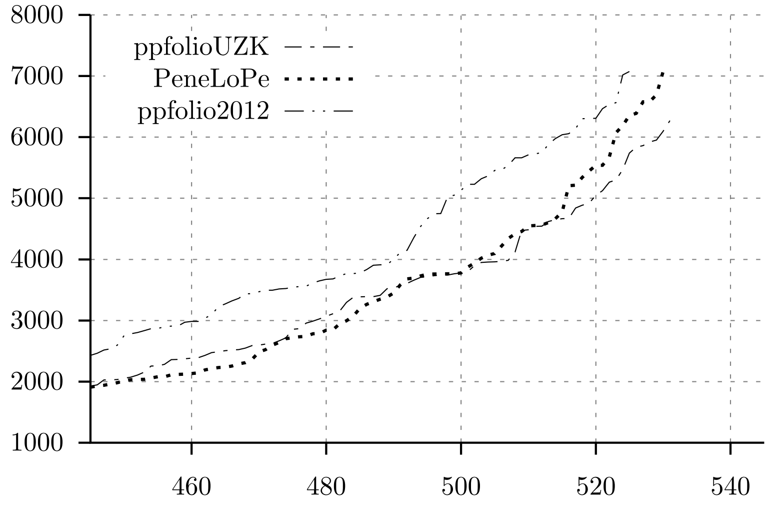
4.7 Independent evaluation

`PeneLoPe` entered the different SAT Challenge/Competitions since it was created in order to have an independent evaluation. The results of the competitions should not be taken as an absolute value of the different solvers. Indeed, the results depend highly on parameters such as the time limit, the selected instances (see page 45),... Moreover, the parallel track in those competitions evolved from single thread competition and compare only the number of instances solved and the average time. This completely ignores the question of the acceleration provided by multi-core. The results are presented in Figure 4.6, 4.7 and 4.8. For each of those competitions, the results shown are those from the solvers reaching the top 3.

4.7.1 SAT Challenge 2012

The SAT Challenge 2012 used 600 instances. Out of the 576 instances solved at least once, 264 are SAT and 312 are UNSAT. The solvers had the possibility to use 8 cores, 12 GB of RAM and a wallclock time limit of 900 seconds. 19 solvers entered the challenge.

`PeneLoPe` solved 530 instances, making the second place at 1 instance from the first position. It is also interesting to note that in the top 3, `PeneLoPe` was the only solver using a single core engine. Indeed, both ppfolio [Roussel, 2012] and pfolioUZK [Wotzlaw et al., 2012] were using different SAT engines (plingeling, sparrow, . . .)



(a) Cactus plot

Solver	SAT	UNSAT	total
`pfolioUZK`	**254**	277	**531**
`PeneLoPe`	240	**290**	530
`ppfolio2012`	241	268	525

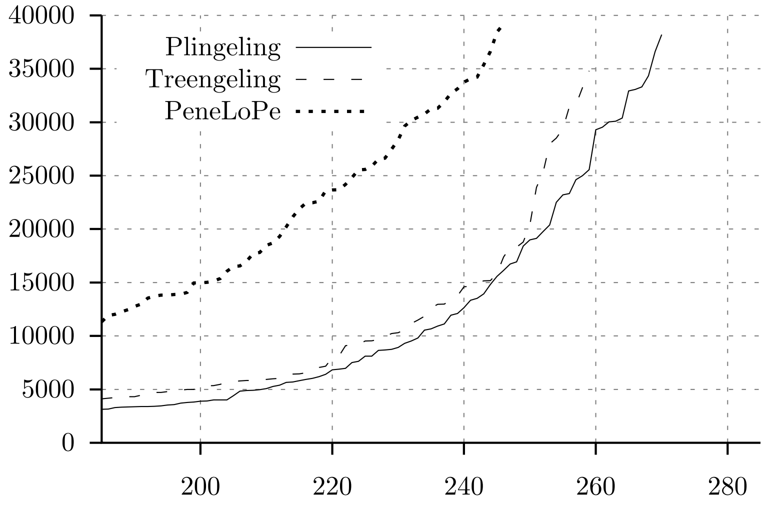
(b) Detailed results

Figure 4.6: SAT Challenge 2012

4.7.2 SAT Competition 2013

The SAT Competition 2013 used 300 instances. Out of the 290 instances solved at least once, 148 are SAT and 142 are UNSAT. From those instances, 90 represented cryptographic problems. 12 solvers entered the competition. The solvers had the possibility to use 8 cores, 15 GB of RAM and a wallclock time limit of 5000 seconds. Multi SAT engines portfolio had their dedicated track.

`PeneLoPe` solved 247 instances, making the third place of the competition. It is interesting to see that Plingeling was able to solve 11 instances more than Treengeling, witch itself solved 13 instances more than `PeneLoPe`.



(a) Cactus plot

Solver	SAT	UNSAT	total
`Plingeling`	**141**	**130**	**271**
`Treengeling`	137	123	260
`PeneLoPe`	133	114	247

(b) Detailed results

Figure 4.7: SAT Competition 2013

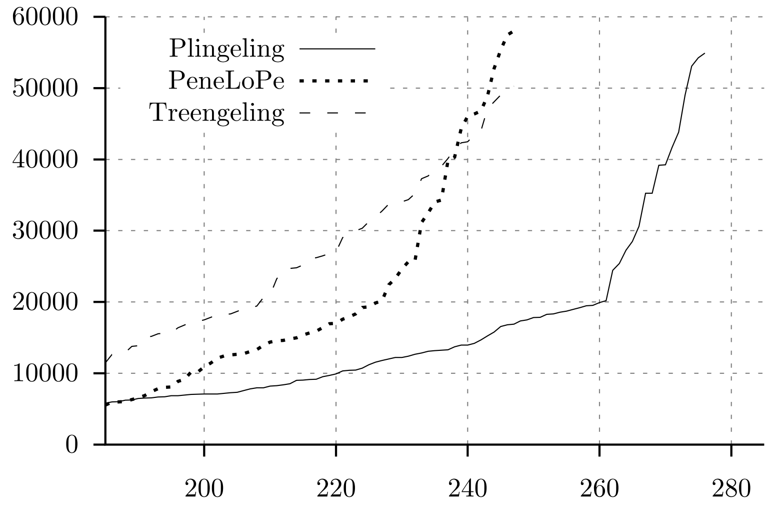
4.7.3 SAT Competition 2014

The SAT Competition 2014 used 300 instances. Out of the 278 instances solved at least once, 128 are SAT and 150 are UNSAT. The solvers had the possibility to use 12 cores, 22 GB of RAM and a wallclock time limit of 5000 seconds. 15 solvers entered the competition.

`PeneLoPe` solved as many instances as Treengeling but with an average running time lower, `PeneLoPe` obtained the second place.

4.8 Conclusion

In this chapter, we proposed the *lbd* metric as a way of evaluating of the quality of a clause for clause exchange. Moreover, by adding the freezing mechanism [Audemard et al., 2011b], we were able to handle more information per thread. The solution proposed is scalable on a multiprocessor machine. However, the number of core/processors on a machine is limited compared to the 'cloud' and grid architectures. Therefore, using such architectures could potentially improve the results.



(a) Cactus plot

Solver	SAT	UNSAT	total
`Plingeling`	**127**	**150**	**277**
`PeneLoPe`	120	128	248
`Treengeling`	113	135	248

(b) Detailed results

Figure 4.8: SAT Competition 2014

CHAPTER 5

Dolius: a distributed framework

Strength through Unity

— Belgian Motto

This chapter presents the contribution that leads to the following publications [Audemard et al., 2014c, Audemard et al., 2014e, Audemard et al., 2013b]

5.1 Introduction

Cloud computing can change the landscape of computer science: it is now possible to request a virtually unlimited number of computing units that can be allocated within a few seconds. In the case of SAT solving, this fact means that larger formulas, much more difficult to solve could be considered, assuming that we dispose of a parallel SAT solver that scales well across different computing units.

Unfortunately, such a scalable solver does not actually exist. Worst, a recent study about parallelization of SAT [Katsirelos et al., 2013] shows that it appears to be very difficult to benefit from portfolio parallelization for modern CDCL solvers. Yet, since the emergence of multi-core CPUs, numerous parallel SAT solvers have been proposed by the community. From the simple script that runs in parallel the best known sequential solvers (see page 55) to complex engines that are able to share knowledge (see page 55 and Chapter 4), most of the (empirically) best attempts are based on the portfolio schema which exhibits *per se* limitations in term of scalability.

The goal of this contribution is twofold: first, propose a framework allowing to facilitate the creation of distributed divide and conquer (D&C) solvers. Those are alternatives to the parallel paradigm that behaves the best in practice for SAT solving: portfolio solvers. Indeed, portfolio techniques monopolize the prizes of each SAT competition, since the parallel track has appeared. By reducing the cost of creating a new distributed solver with our framework, D&C can possibly challenge this position. The second goal is to contribute to improve scalability in parallel solutions for SAT. The chapter is organized as follows: in the next Section, we present the differences between the main schemes to parallelize SAT solvers and their implications. In

Section 5.3 we present the main features of our framework called `dolius`. Its API, presented in Section 5.5, enables any SAT solver to be easily plugged to it. Next, we evaluate in Section 5.6 the efficiency of our framework when instanced with one of the best current sequential solvers, and we finally conclude with some perspectives.

5.2 Pathological Cases of D&C

Let ϕ be a CNF formula. $\Sigma = ((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ is also a CNF formula. Dividing the search on either a or b causes some problems.

5.2.1 The Ping Pong Effect

If the search on Σ is divided using a, one of the subsequent task is very light, since it is easy to prove that $\Sigma \vDash a$. Hence, just using unit propagation, it is possible to show that $\Sigma \wedge (\neg a) \vDash \bot$. The slave that receives such (sub)-formula can prove it inconsistent without any exploration at all, and asks again for work very quickly. This is a problem, since work division has a cost, particularly because of network communication.

If bad choices are successively made when dividing the CNF, then one of the worker repetitively receives a trivial subproblem, and spends more time asking for work than actually solving the problem. This phenomenon is called *Ping-Pong effect* in earlier work [Jurkowiak et al., 2001].

5.2.2 Useless Division

Back to our example. If the search is divided on b, then each slave actually works on the same formula: $a \wedge \phi$. This is clearly not ideal, since redundant work has to be avoided as much as possible.

Hence, in such a situation, it would be desirable to divide the search with respect to a variable from $Var(\phi)$ rather than either a or b. Those results pled for a careful analysis of the division strategy.

5.2.3 Towards Scalability

Portfolio strategies are efficient on multicore architectures, but find their limits when they are used with a large number of computing resources. Indeed, adding more and more resources is not helpful for this kind of approach. This just leads to a large amount of redundant work, since in practice, the same parts of the search space are very often explored by several workers simultaneously. Moreover, a recent study shows that portfolio is a resolution schema that exhibits *per se* clear efficiency limitations [Katsirelos et al., 2013].

On the contrary, adding more resources benefits better to the D&C framework, providing that useless divisions are not regularly achieved. In the next Section, we formally present our divide and conquer SAT solver, called `dolius`.

5.3 dolius

Since our goal is to contribute to improve scalability in parallel solutions and to deploy such solution on distributed architectures, it appears better suited to propose a novel approach based on the divide and conquer paradigm. This is the main objective of our framework `dolius`. As presented in the next section, our framework can easily plug any available SAT solver using a simple API and can be extended as a portfolio SAT solver. Let us start with the main architecture of `dolius`.

`dolius` uses one master and many slaves. This architecture was chosen for different reasons: first of all, it allows a much easier development. Such an architecture is used by webservers, which can handle ten thousands of concurrent connections. As depicted later, the work of the master in `dolius` is very light, and only consists in putting in touch hungry slaves with active ones. Therefore, as our master's task is lighter than one of those webservers, such an architecture appears appropriate.

Even so, in contrast to full-decentralized techniques, this approach can clearly lead to bottlenecks. If this case would occurs, a tree structure could be implemented (a slave could be composed of a `dolius` master that uses its own sub-slaves), as the one provided with the Domain Name Server (DNS) system.

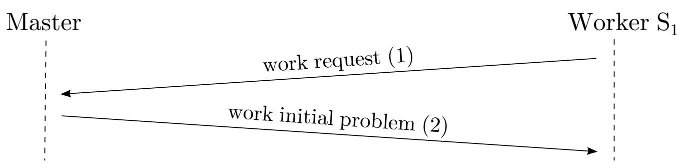
Each slave is a SAT solver, whereas the master is a process that does not participate actively to the search, and is only used as the cornerstone for communication between the slaves. The master knows all its active slaves (the latter ones contact the former one in order to register) but slaves do not know each other. Moreover, the master is designed to be flexible, and workers can be added on the fly during the search. To divide the work, `dolius` allows a divide and conquer approach through guiding-paths. Such guiding paths are not reduced to a single variable but can also split the formula with two sets of clauses. However, in that case, one needs to be careful on some properties that the sets of clauses must verify.

5.3.1 Communications

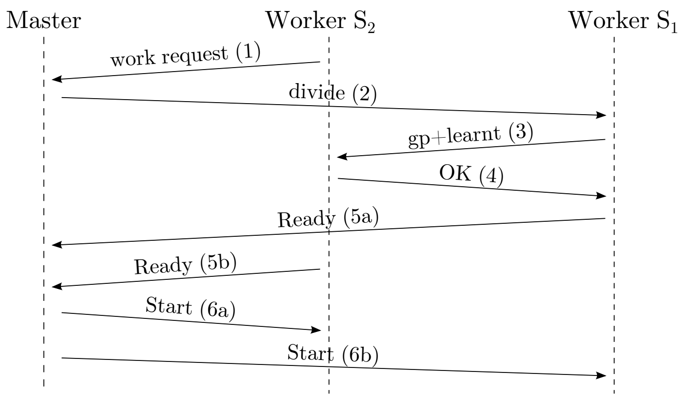
Work request

Let us review the different communication phases that occur within `dolius`. First, a worker w_i needs to request some work to the master. From the master's perspective, two possibilities can appear: either the worker is the first one to request some work or there are some active slaves. If w_i is the first one, the master will send a message to let w_i know that he must work on the initial problem. This message exchange is depicted in Figure 5.1(a).

If w_i is not the first slave, the master needs to choose an active slave w_a to ask him to divide its load. Different criteria can be considered to choose w_a. We propose the following work-stealing scheduling strategy: the master node stores a FIFO data structure of currently working nodes, proposes the first node to balance its load and puts this node at the end of the list. One execution of this strategy is depicted at Figure 5.2. This system allows to ensure that work request is sent fairly between active workers. In addition, this choice has been made to avoid contacting the same active slave several times in a row in case of simultaneous requests for work to the master. A work request can be denied by the active worker if the underlying solver has not worked enough.

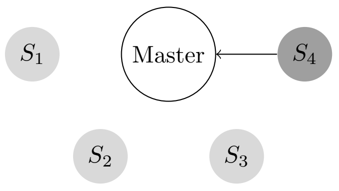


(a) 1st worker

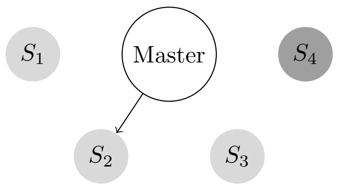


(b) other workers

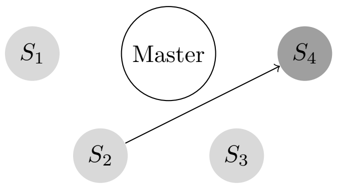
Figure 5.1: Communications to initialize workers



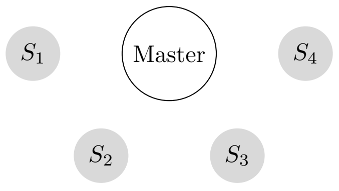
(a) Slave S_4 finishes its load, so it contacts the master



(b) Master asks to the slave S_2 if it accepts to divide it load. S_2 is choosen as it is on top of the master's FIFO containing every active slaves.



(c) If S_2 accepts, it sends a part of its load to S_4



(d) All slave are now active. S_2 and S_4 are now at the bottom of the fifo containing the slaves

Figure 5.2: Load Balance through work stealing procedure of `dolius`

In order to be as opportunistic as possible with respect to available resources, a load balancing technique must then be implemented. Indeed, in practice, certain slaves finish their task before the others and become idle. There exists two main schemas to achieve this load balance. With the first one, each busy worker regularly looks up in some defined "neighbourhood" of workers to (possibly) find an idle worker and *push* some work towards it. The second schema makes responsible idle workers to contact an active worker in order to *steal* a part of its workload.

When a worker has to divide its workload, it is also possible to send the learnt clauses or a selected subset. The choice of sending the clauses, and which clauses to send is somewhat important as some clauses will not be used during the next search, but others might be. Moreover, as communication is not done through shared memory, the cost is not negligible. The communication is made through TCP/IP, minimizing the need for external API. This choice was mainly made for portability and reliability of communications.

In between information exchange

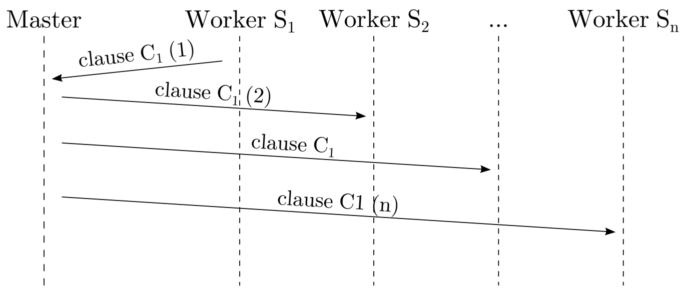
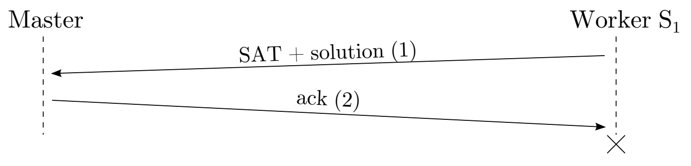


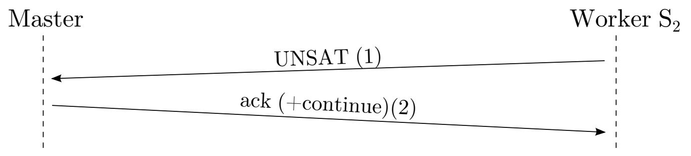
Figure 5.3: Clause multicast

In order to achieve good performances and reduce the effect of an eventual bad division [Audemard et al., 2014e], solvers can choose to send to each other (through the master) some clauses learnt during the search. This induces that the solver can also find which part of its guiding path is responsible for the UNSAT answer and send it to the other workers. If there is no guilty part in the guiding path, an empty clause can be sent to stop all other workers. This is extremely useful as it can balance a bad division. To understand the opportunity of this mechanism, let us suppose that a worker W has the guiding path $\mathcal{G} = l_1 \wedge ... \wedge l_n$. It is possible that W generates the clause $\mathcal{C} = \neg l_1 \vee \neg l_2$ during its search. This clause can be very useful as it leads to the termination of every active node using $l_1 \wedge l_2$ as part of their guiding path. Furthermore, if a worker is able to generate an UNSAT proof without using its guiding path, an empty clause can be sent in order to stop every other worker.

When solving an UNSAT instance with a portfolio approach, the process stops after the first thread has stopped. In classical divide and conquer, the answer can be given only when every sub-problems have been found UNSAT. Therefore, sending the responsible part of the guiding path can bring us back to the portfolio situation, since any worker can possibly prove the original CNF unsatisfiable.



(a) SAT answer



(b) UNSAT answer

Figure 5.4: Communication when a worker has ended its search

End of search

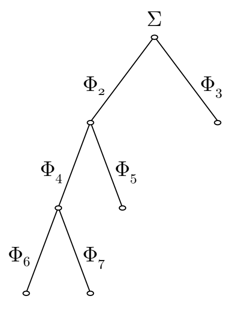
When a worker has ended its search, the master must be notified of it. If the answer found for the sub-space is SAT, the solution must be provided as shown in Figure 5.4(b). This will lead the master to send a stop message to all of its active slaves.

If the answer found in UNSAT, the master can not deduce anything for other workers. Only if the worker providing the answer is the last working slave, can the master stop as the instance has been proven UNSAT.

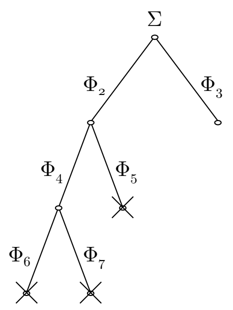
5.4 Incremental guiding path

As a solver is needed to test the platform, we have modified `glucose`. The work division strategy implemented in `createGP` is based on unit clauses where the chosen literal is one of the literal with the highest *VSIDS* value when the work is divided and must also provide a good balance value between the two branches. The balance value is obtained by using look-ahead techniques. As we are using unit clauses for the guiding path, we incorporated them in the `assumption` vector (see page 43) of the solver, as presented done in [Hyvärinen et al., 2011]. This provides us with more information whenever we find UNSAT. Indeed, through the `analyzeFinal` function that was designed in `MiniSat`, we are able to find the responsible part of the guiding path, if any. Once found, this information can be sent to other solver (through the master) to avoid exploring redundant search spaces. An example is provided at Figure 5.5. Let us suppose that there are 4 active workers working on the following part of the search space: $\Sigma \wedge \Phi_3$, $\Sigma \wedge \Phi_2 \wedge \Phi_5$, $\Sigma \wedge \Phi_2 \wedge \Phi_4 \wedge \Phi_6$ and $\Sigma \wedge \Phi_2 \wedge \Phi_4 \wedge \Phi_7$. If one of the slave working on a subspace of $\Sigma \wedge Phi_2$ finds UNSAT and Φ_2 as reason, the negation ($\neg\Phi_2$) can be sent to every active slave. This message will not provide any more information to the slave working on $\Sigma \wedge \Phi_3$. However, it will effectively stop the other slaves, allowing the system to concentrate the effort on another search space.

Such technique can reduce the effect of a bad division as the resources involved with an easy search space will be stopped and will automatically use another one. Using `assumption` also allows us to keep clauses whenever the guiding path is changed, allowing us to keep clauses that were generated.



(a) Before $\neg\Phi_2$ is found, 4 active workers



(b) After $\neg\Phi_2$ is found, 1 remainding active worker

Figure 5.5: Before and after $\neg\Phi_2$ has been sent to every worker

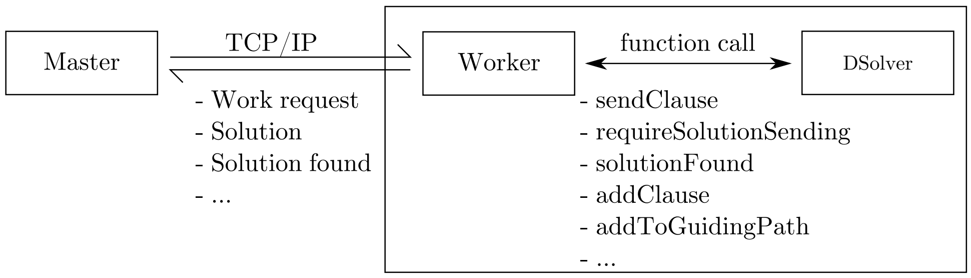


Figure 5.6: Communication between the master, the worker and the solver

5.5 Application Programming Interface

`dolius` has a clear separation between the main platform (master, slaves, communication...) and the SAT solver as shown in Figure 5.6. This offers multiple advantages. The main one is that each worker does not need to use the same SAT solver (`MiniSat` [Eén and Sörensson, 2004], `PeneLoPe` [Audemard et al., 2012b], ...) allowing to easily introduce a portfolio approach inside the divide and conquer paradigm. This clear separation is possible with the following API.

This API has actually been designed to be as complete as possible, but all functions are not needed, some of them are optional. First, we provide the minimal set of functionalities that have to be implemented to make use of `dolius`.

To make a solver distributed with `dolius`, only a few functions have to be fully implemented. Those functions are summarized in Table 5.1 that considers the life of a SAT search as 3 main phases: its start/initialization, the actual search period, and its end of the search.

The implementation of some of those functions should not be hard. For instance, the functions dedicated to retrieve information about the status of the search are easy to implement: `solutionFound()` is a simple state function that is used to know whether the search is still active, or if a solution has been already obtained. `isSolutionFoundSAT()` is used to know the nature of the delivered answer (SAT / UNSAT), whereas the function `getSolutionLiteral()` is called to get the model, when a SAT answer has been obtained.

```
//initialization
void setCNFFile(const char* inputFile);
void initialize(int nbVar, int nbClauses);
//thread related functions
void run();
void stop();
//clause database modification
void addLearntClause(const std::vector<int>& clause);
void addClause(const std::vector<int>& clause);
//iterators
void learntClauseIteratorRestart();
void learntClauseIteratorNext(std::vector<int>& clause);
void guidingPathIteratorRestart();
void guidingPathIteratorNext(std::vector<int>& clause);
int  getGuidingPathSize() const;
//guiding path modificators
bool createGuidingPath(std::vector<std::vector<int> >& gpA,
                       std::vector<std::vector<int> >& gpB);
void addToGuidingPath(const std::vector<int>& clauses);
//sat related information
bool solutionFound() const;
bool isSolutionFoundSAT() const;
int  getNbVar() const;
int  getSolutionLiteral(int var) const;
int  getNbLearntClauses() const;
```

Figure 5.7: functions to implement in order to incorporate a solver in `dolius`

	start	**search**	**end**
mandatory	`initialization` `set_GP`	`run(), stop()` `create_GP`	`sat_related_info`
optional	`addLearntClause()`	`addClause()` `iterators`	

Table 5.1: Summary of functionalities used by the `dolius` API

In the start phase, `initialize()` serves to allocate memory, and the input CNF file can be read through `setCNFFile()`, preparing the solver to be runned. In the "search" phase, the function `run()` is called to actually start the search process in its dedicated thread, whereas `stop()` is on the contrary used to interrupt this process.

Only 2 functions need more code to distribute a solver: `addToGuidingPath()` is used to restrain the search space of a worker, with respect to a guiding path given in parameter. As the guiding path is provided as a set of clauses, it allows developers to create new heuristics to divide the work. The other one, `createGuidingPath()`, represents the heart of load balancing, since it is called when an *idle* worker makes a work-steal to an active one. Thus, the active solver has to divide its own task to give a part of it to the active solver. In our current implementation, the division is made using a unit clause selected through a look-ahead procedure (see [Audemard et al., 2014e]), but other division strategies can be implemented in this `createGuidingPath()` function.

Iterators on learnt clauses are also used to send those on work division. By tweaking which clauses are iterated on, a heuristic can be implemented to choose how many clauses are sent when the work is divided. Those will be added by the *idle* worker through `addLearntClause()`.

In addition, the API proposes numerous optional functions (e.g. iterators on the guiding path, `getGuidingPathSize()` , etc.) that can be implemented for statistical and debug purposes. Hence, our API has been designed to be both complete and easy-to-implement. As an example, the code needed to plug `glucose` [Audemard and Simon, 2009a] with `dolius` consists of less than 300 lines of code.

Let us also note that a solver integrated in `dolius` has also access to functionalities of our framework (log files, etc.). And in order to ease the integration of solvers, several tools have been developed such as a graphical depiction of the resulting guiding tree that can be animated and colouration of the log files.[1]

5.6 Evaluation

Two sets of evaluations were made, one focused on the scalability of the approach, and the second using some standard SAT evaluation technique.

[1]The results of those tools are presented at `http://www.cril.fr/~hoessen/dolius.html`

5.6.1 First evaluation

We have experimentally tested `dolius`, wherein each slave uses a modified version of the well known CDCL solver `MiniSat` [Eén and Sörensson, 2003]. The guiding path is added as a set of initial clauses. Therefore, when a new request is being made, the solver has to remove all its data and reinitialize itself. As guiding path selection, we take as literal one of those with the highest VSIDS value. Once we obtain the set of literal with high VSIDS, the distinction is being made by using a look-ahead technique to obtain one with a balanced search space. As for the clauses sent from one node to another, every learnt clause is sent.

The experimentations of this first evaluation have been conducted on different computing units that exhibit the exact same features: dual socket `Intel XEON X5550` quad-core 2.66 GHz with 8 MB of cache and a RAM limit of 32GB, under Linux CentOS 6 (kernel 2.6.32). All machines are linked through a HP ProCurve 4108gl switch using a gigabit connection, allowing each node to communicate with any other node with a maximum speed of 1Gbps.

The timeout is set to 1200 seconds wall clock (WC) for each instance. If no answer is delivered within this amount of time, the instance is considered unsolved. We use the instances from the application track of the 2011 SAT Competition. The measured time comes from the moment the master was started up to the moment that the last node receives a stop signal. Each test is performed five times.

First of all, over the 300 instances, 141 were solved at least one time. Table 5.2 provides some general information. It shows for each number of worker, how many instances are solved 0, 1, 2, 3, 4 or 5 times. The most significant evolution is the number of instance unsolved. Using only 1 worker, there are 48 of such instances. However, using 16 workers there are only nine instances of the 141 that are unsolved. This result shows clearly that dividing the instance is really beneficial as we were able to solve instances that are not solved using only one worker.

nb workers	0 time	1 time	2 times	3 times	4 times	5 times
1	48	1	4	1	0	87
2	35	5	4	5	8	84
4	26	8	6	5	7	89
8	22	9	6	6	8	90
16	9	6	12	8	20	86

Table 5.2: The number of instances that were resolved a given time by a given number of workers.

More precisely, the results can be divided in three categories. The first, about 54% (39 SAT, 38 UNSAT), consist of instances where a speedup can be observed. Out of this categories are the instance depicted at Figure 5.8 and 5.9.

As we can see, the speedup for instance depicted at Figure 5.8 is 1.94 for two workers compared to one, 1.67 for four workers compared to one and continues to decrease as the number of workers increases. As for the network, there are no exponential use of it. With four workers, we use 2.29 times more network than with two workers. With eight workers, we use 1.77 times more than with four workers and the factor continue to decrease as the number of workers increases.

instance	SAT ?	**1 worker**		**2 worker**		**4 worker**		**8 worker**		**16 worker**	
		time	div	time	div	time	div	time	div	time	div
`AProVE07-21`	UNSAT	415	0	213	3	127	11	84	28	**65**	47
`md5_48_3`	SAT	931	0	471	1	294	3	220	9	**219**	19
`rand_net60-40-10.shuffled`	UNSAT	—	—	753	6	301	22	146	47	**101**	97
`vmpc_36.renamed-as.sat05-1922`	SAT	—	—	—	—	—	—	668	13	**214**	40
`rand_net60-30-1.shuffled`	UNSAT	**26**	0	**26**	0	**26**	0	**27**	0	**27**	0
`hsat_vc12062`	UNSAT	**37**	0	62	7	139	55	176	145	218	348

Table 5.3: Average running time in seconds on 5 runs for a selection of instances, with an increasing number of workers

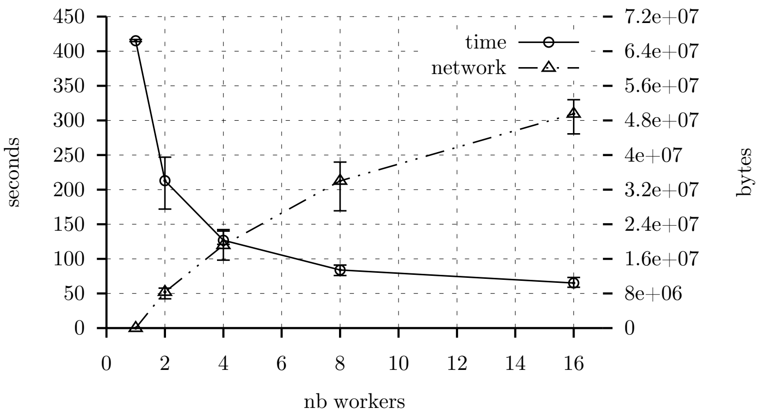


Figure 5.8: Instance: AProVE07-21. Number of variables: 3189, number of clauses: 11039, UNSAT

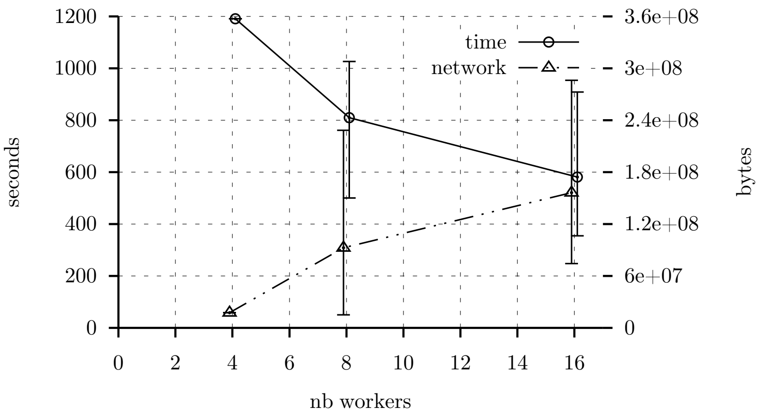


Figure 5.9: Instance: UCG-15-10p1. Number of variables: 200003, number of clauses: 1019221, SAT

The first element that we have to point out for instance depicted at Figure 5.9 is that using less than 4 workers, no answer is found under 1200 seconds. The speedup for eight workers compared to four is 1.47, and 1.39 using sixteen workers compared to eigth. As for the network, with eight workers, we use 5.26 times more network than with four workers. With sixteen workers, we use 1.68 times more than with eight workers.

The second category consists of about 31% (29 SAT, 15 UNSAT). For those instance, generally solved quite fast, our approach does not provide any speedup. This can be explained by the fact that time is needed in order to activate every node, read the instance file, splitting the work, etc. However, as we are trying to provide a new framework to solve difficult instances, this category is not the most important to us.

The last category is where the problems are. They are about 14% (9 SAT, 11 UNSAT). Instances in this category tend to take more time as the number of resources increase. This is partially due to the fact that on this set of instances ping pong effects are still observed.

Finally, Table 5.3 provides some details for representing instances. For different number of workers it gives the average running time (or "—" if timeout is reached for all 5 runs) and the average number of work divisions. The first four instances come from the category where a speedup is observed. Note for example that `vmpc` instance can be solved only using 16 workers. In such instances, using only 2 workers leads to too few work divisions, here the divide and conquer approach seems to be the good choice. The instance `rand_net60-30-1.shuffled` is a typical too easy instance: there are no division. The last instance provides an example where increasing the number of workers leads to increase the running time. Note that with only 1 worker the instance is solved in 37 seconds and (of course) without divisions and adding workers increases (obviously) the number of divisions but decreases the running time. We suppose that this is a typical ping-pong effect.

5.6.2 Second evaluation

In the following, the hardware used is 2 Dell R910 with 4 Intel Xeon X7550 providing each 8 cores making 32 cores available per node. Each node has a gigabit ethernet controller and 256GB of RAM. The installed operating system is CentOS 6. For each experiment, given an instance and a number of workers, we choose a time limit of 20 minutes. Let us also note that we consider *wall clock* time, instead of CPU time, in this Section. We only made one run for a given instance and a given number of workers, because of limited resources. Indeed, one run, for a given number of workers, can make use of 17 hours in the worst case (20 minutes $\times$ 51 instances). As we consider 7 numbers of workers, with and without clause sharing, one entire run lasts more than one week. Running 10 times each instance would have been computationally very expensive.

In order to evaluate our platform, two elements are needed: instances and a solver that will be plugged in. Concerning instances, as the resources needed to make tests with many slaves may be quite important, a subset of the instances from the SAT Competition 2013 (application track) [Balint et al., 2013] are used. To be used, an instance needs to be solved by at least one of the five first parallel solver form the SAT Competition 2013 and may not be solved by everyone of them. Those criteria insure us to evaluate ourselves against *reachable* instances. From that set, we skim large instance families. We reduced that set to have a manageable number of

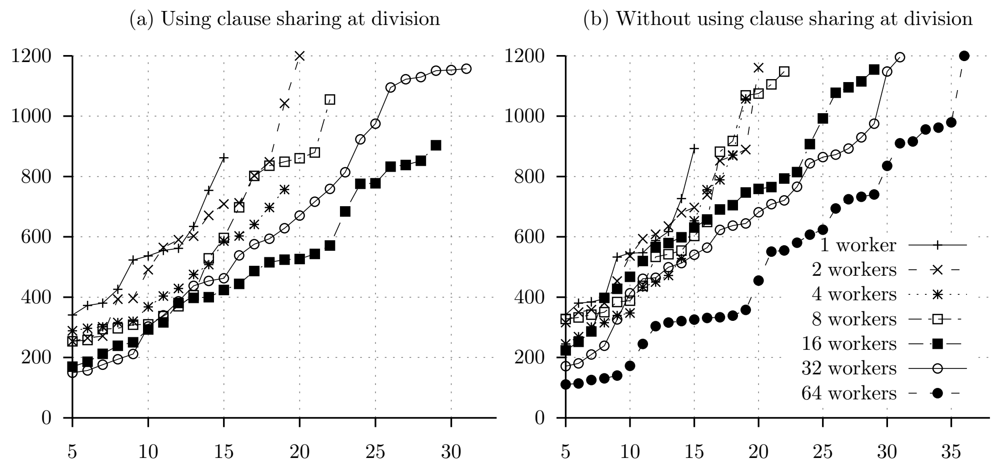


Figure 5.10: Scalability of the `dolius` + `glucose` implementation

instances: 51. In the final set, there are more unsatisfiable instances (33) than satisfiable ones (18).[2]

As solver, two flavors of the modified `glucose` with incremental guiding path (see Section 5.4) were tested. In the first one the learnt clause iterator send the learnt clauses under the condition that their *literal block distance (LBD)* value needs to be lower than 4 and the maximum number of clauses sent must be lower than 10% of the total amount of learnt clauses. The results of this version of `dolius` +`glucose` are shown in Figure 5.10 (a). As we can see, the cactus plot that represents 1 worker is higher than other curves showing that the work division helps in reducing the overall solving time. However, the curve for 16 workers is lower than the one using 32 workers, for a equivalent number of solved instances.[3] The main reason for this is the communication cost.

The second flavor is obtained by simply deactivate the communication of learnt clauses at division through a different implementation of the learnt clause iterator. The results of this version are shown in Figure 5.10 (b). As we can see, in this version, the 32 workers curve is lower than the 16 workers curve. This motivated us to try this flavor using 64 workers. The gain obtained is quite significant, the curve being always lower than any other curve and providing answer for 5 more instances for a grand total of 37 solved instances.

Both flavors have their limitation, communication providing better results with a low number of resources and no communication with a higher number of resources. This should mean that in order to develop a good heuristic concerning the communication of learnt clause, the number of workers should be taken into account, in order to avoid the case where the learnt clause communication is completely disabled. This appears essential to keep an effective scalable communication when a large number of workers is involved. The experiments underlines that new heuristics that are able to take into consideration this increase of computing units have to be designed.

In order to understand those results and compare those with the current *state-of-the-art*, let us compare to 3 shared memory solvers among the best ones proposed in the SAT Competition

[2]The exhaustive list can be found at `http://www.cril.fr/~hoessen/dolius.html`

[3]Using 32 workers, we were able to solve 2 instances more but due to the non determinism, solving 2 instances more is not really significant.

Solver	**#threads**	**SAT**	**UNSAT**	**Total**
`PCASSO`	32	9	21	30
`plingeling`	32	14	23	37
`PeneLoPe`	32	16	26	42
`dolius` + `glucose`	32	9	23	32
`dolius` + `glucose`	64	13	24	37

Table 5.4: Results for some of the parallel SAT solvers submitted to the SAT Competition 2013 and `dolius` + `glucose`

2013: `plingeling` [Biere, 2013], `PCASSO` [Irfan et al., 2013] and `PeneLoPe` [Audemard et al., 2013a]. They were choosen for the following reasons: `plingeling` was the 2013 winner, `PCASSO` uses a division strategy and the authors wanted to compare against themselves with our previous solver, `PeneLoPe`. Each solver is launched using 32 threads as it is the highest number of cores available on a single machine. Results are shown in Table 5.4. First, we have to recall that those solvers use shared memory for communication making their communication *de facto* faster than the one proposed in `dolius`. Nevertheless, shared memory are a lot less scalable than distributed ones. Moreover, `PeneLoPe` and `plingeling` are portfolio: they use different configurations in each thread, allowing to increase the orthogonality of the search. And with a greater orthogonality comes a more robust solver against different families of benchmarks. With those details in mind, we can see that using twice the resources of `plingeling` –the solver who won the SAT Competition 2013–, we are able to obtain the same number of instances solved. Finally, `PCASSO` is partitioning the search space iteratively, an approach similar to the one described here. We can see that we achieve similar results with the same amount of threads/workers.

5.7 Evolutions

As the API was proposed to be somewhat flexible, we discuss some possibilities to re-implement some techniques using our framework. Solvers could send an empty guiding path to obtain a portfolio. As the API provide a way to send a clause to every other known worker, it is possible that the portfolio communicates its learnt clauses. Moreover, if different solvers are used, it allows to create a portfolio *à-la* ppfolio but with communication.

A second way to use the `dolius` platform would be by using using an approach as described in [Hyvärinen et al., 2011] (see page 58). In this approach, when creating the guiding tree, workers are not only working on the leaves of the tree, but on every node. This system can be done using the following mechanism, depicted at Figure 5.11 When a worker w receive a work division request, the worker send the guiding path $\mathcal{G}_w \wedge l$, where $\mathcal{G}_w$ is its current guiding path. w keeps $\mathcal{G}_w$ as guiding path. When w receive a second work division request, it send the guiding path $\mathcal{G}_w \wedge \neg l$ and keeps $\mathcal{G}_w$ as guiding path and denies any further division until $\mathcal{G} \wedge \Sigma$ is solved. This resolution of $\mathcal{G} \wedge \Sigma$ can be done through the resolution of $\mathcal{G}_w \wedge l$ and $\mathcal{G}_w \wedge \neg l$, or by the worker w directly.

A third way to use the `dolius` platform would be by using pre-computed guiding path as

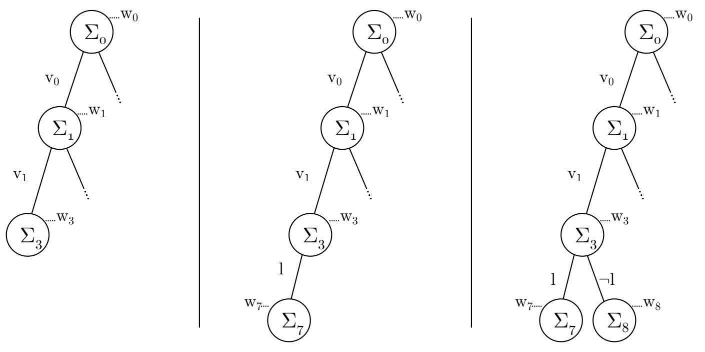


Figure 5.11: Steps to divide the work in `dolius` as described in [Hyvärinen et al., 2011]. Left is the initial guiding tree, with each node related to a given worker w_i. The centre figure represent the tree after w_3 divided a part of its work to w_7. Right represent the guiding tree after w_3 divided for a second time its work, towards w_8 this time.

described in [Semenov and Zaikin, 2013]. At the i-th division, the requesting worker would receive the i-th pre-computed guiding path. However, in order to implement this, an additional service would be needed to known the value of i as the current implementation of `dolius` does not provide such information.

Finally, as last example of how `dolius` can be used, is by using as worker a parallel solver such as `PeneLoPe` or syrup [Audemard and Simon, 2014]. This way, network communication would be handled by `dolius` whereas communication on the same host would be handled by those solver using a more efficient communication channel.

5.8 Conclusion

In this contribution, we provided a simple interface to create distributed and parallel solvers. As the requirement are as low as possible, it can be used to create portfolio as well as divide and conquer solvers. The framework can be easily extended and offer multiple possibilities to the scientific community. Moreover, we implemented two different versions that provided some promising results. Indeed, those are shown to be scalable in more than half of the solved instances and competitive compared to some state of the art techniques.

CHAPTER 6

Conclusion

The end has no end.

—Julian Casablancas

In this thesis, we depicted different ways to solve the Boolean satisfiability problem using the parallel paradigm. Different points can be taken from this study. First and foremost, we saw that some techniques such as the portfolio seem simple, or might even been seen as simplistic. Those work really well, adding even more credit to the saying “keep it simple stupid”. However, we saw that it is not as “stupid” as it seems, as we can provide explanation of such results: the principle that “there is no free launch” is playing in favour of portfolio. Moreover, portfolio are able to gain directly from the performances of their underlying solver core.

Second, we were able to produce two different scalable methods. The first one, portfolio have been studied for over five years and are able to produce descent work thanks through shared memory. Those, combined with communication are able to scale without having to introduce different configurations. The second, divide and conquer has been studied for several decades. However, most of the proposed approaches were not generic enough. Either too much dependency on the hardware, or not enough freedom in methodology for the division. This was addressed with our second contribution.

6.1 Perspectives

Doing a thesis is only the beginning in term of scientific research, it is only a step in a long journey. Once a step is done, we must look towards the destination of the next steps. Different possibilities are available after the work presented here. There are still plenty of research that could be done in the divide and conquer part. Trying to find better heuristics, studying the gain of diversification and trying to define an acceptable work load. Other techniques could be studied such as the possibility to restart the search without loosing information. We did concentrate our efforts on the solving part but usually, the instances are pre-processed. Those pre-processors, could also use the parallel paradigm.

The different techniques that were considered in this thesis use the resolution rule to solve the instances. However, other techniques could be used to solve the Boolean satisfiability problem where some of those might be more suitable for the parallel paradigm. Trying to find a new technique could reach better results than those presented here. But, there are different challenges that should be addressed in order to be competitive. First, modern CDCL solvers are the results of years of improvement. A new method should be either powerful enough to compensate those years of research, or it should be very well implemented (in terms of cache access, memory layout, . . .). The second drawback of looking in another direction is that the gain might be higher, but the risks are also higher. Therefore, before entering such unknown realms, one must be ready to take those risks in our modern scientific world.

Finally, let us quote the movie Jurassic Park: "*your scientists were so preoccupied with whether or not they could that they didn't stop to think if they should*". Indeed, we may not be separated from the consequences of our work and in our world where the power comes at a price, it is interesting to take a step back and think about the ecological consequence of this thesis. The cumulated CPU time for the multiple tests lay beyond a decency. However, the propositions made can improve the efficiency of the users. By providing results faster, it can also lead to a better integration of verification techniques and spread its usage. This can later have many implication such as:

- less security update needed for software, leading to less power consume for the propagation and installation of those updates
- improved security in the software and hardware of transportation (cars, planes, buses, . . .), leading to avoiding deaths due to those
- providing better solutions and more energy efficient solutions to problems
- . . .

Therefore, the author of this thesis is convinced that those tests were worth it.

CHAPTER 7

French resumé

En peu de temps parfois on fait
bien du chemin.
—Jean-Baptiste Poquelin

La logique est définie comme étant l'art du raisonnement de son expression. Il est très intéressant de noter que la logique est au carrefour des mathématiques, de la philosophie ainsi que de l'informatique. L'impact de la logique sur l'informatique est énorme au travers de la logique booléenne. Tout les circuits imprimés actuels utilisent la logique pour décrire les différentes opérations qui doivent être effectuées.

De nos jours, la logique se trouve un nouvel usage en informatique au travers de la validation automatique de code, de circuits électroniques, ... Ces validations sont effectuées par des logiciels automatisant le raisonnement appelés solveurs ou prouveurs, et trouvent leur utilité bien au delà de l'informatique. Il est possible de trouver ces prouveurs en biologie, mathématique, économie et bien d'autres. Ces solveurs fonctionnent habituellement par la recherche systématique d'une réponse à un problème combinatoire. Une des spécificité des problèmes combinatoire est, selon l'état de l'art de nos connaissances, qu'une réponse ne peut être trouvée qu'en énumérant l'ensemble des solutions potentielles. Et une telle énumération peut être très couteuse en temps. Le problème de satisfaction de formules booléenne sous forme de CNF (SAT) est un de ces problèmes. Celui-ci est très intéressant pour de multiples raisons. Premièrement, n'importe quel problème combinatoire peut être exprimé au travers du problème SAT. Deuxièmement, le problème peut être facilement décrit et repose sur des opérations logiques. Et comme dit précédemment, la logique est fortement étudiée et propose de ce fait moult études sur la résolution de ce problème. Cependant, même à l'aide de ces techniques, une énumération est toujours nécessaire. Il est donc nécessaire de proposer de nouveaux algorithmes pour réduire le temps nécessaire au calcul.

En cherchant de nouveaux algorithmes permettant d'obtenir une solution le plus rapidement possible, certains scientifiques se sont tournés vers le calcul parallèle. Ce paradigme permet l'exécution de plusieurs opérations en même temps. Ceci peut-être obtenu au travers de l'utilisation de plusieurs ordinateurs ou d'ordinateurs multi-cœurs. Heureusement, de telles architectures deviennent la norme pour les ordinateurs personnels jusqu'au téléphones portables haut de gamme. Cela permet donc un accès plus facile au matériel nécessaire.

En plus du matériel, des logiciels sont également nécessaires. Bien évidemment, la programmation parallèle n'est pas la panacée, mais celle-ci peut fournir de nouveaux concepts utile à la recherche. De plus, avec l'augmentation du nombre de problèmes pratiques résolus au travers SAT, fournir des résultats plus rapidement peut être avantageux pour les utilisateurs des prouveurs SAT.

Il existe deux grandes familles d'algorithmes pour résoudre SAT en parallèle partant du problème que le nombre de solutions potentielles est trop élevé. La première est l'approche de type portfolio dans laquelle différents solveurs sont lancés en même temps. De ce fait, en utilisant de mutliples solveurs sur le même problème, ceux-ci n'énumereront pas dans le même ordre et permettront ainsi d'obtenir une réponse plus rapidement.

La seconde approche est la famille d'algorithmes du type diviser pour régner. Pour ceux-ci, le nombre de solutions potentielles est trop élevé pour accepter qu'une solution soit explorée plus d'une fois. Pour ce faire, la formule est divisée en deux sous formules et chacune est donnée à un solveur SAT.

Le but de cette thèse est de proposer deux approches pour résoudre SAT en utilisant la programmation parallèle. La première, `PeneLoPe` utilise l'approche de type portfolio. Par l'utilisation et la combinaison de techniques provenant de l'état de l'art, nous avons proposé un solveur SAT ayant obtenu de bons résultats. La seconde, `dolius` est une plateforme dédiée à la résolution distribuée du problème SAT. Son but est de faciliter la création de solveurs distribués tout en offrant une grande flexibilité sur les critères de division du travail.

Ce chapitre est composé comme suit: la première section décrit brièvement le problème SAT. La seconde section présente les méthodes utilisées pour paralléliser la résolution de SAT. La troisième section présente la première contribution de cette thèse: `PeneLoPe`. La quatrième section présente la seconde contribution de cette thèse: `dolius`. Finalement, la dernière section présente différente perspectives.

7.1 Problème SAT

Nous commençons ce résumé par l'introduction des définitions. Soit une formule booléenne Σ composée des variables booléennes a, b, c. Il est possible de ré-écrire cette formule Σ de manière à ce qu'elle respecte la forme normale conjonctive. Une formule en forme normale conjonctive est composée de conjonction de clauses: $C_1 \wedge C_2 \wedge \ldots \wedge C_n$. Une clause est une disjonction de littéraux $l_1 \vee l_2 \vee \ldots \vee l_n$. Le littéral positif de variable a est a. Le littéral négatif de variable a est $\neg a$.

Soit deux clauses $C_1 = l_1 \vee \ldots \vee l_n \vee a$ et $C_2 = l_{n+1} \vee \ldots \vee l_{n+m} \vee \neg a$. La résolvante de la clause C_1 et C_2, que l'on désignera en utilisant l'opérateur $\otimes_{\mathcal{R}}$, vaut $l_1 \vee \ldots \vee l_n \vee l_{n+1} \vee \ldots \vee l_{n+m}$.

Chaque variable étant booléenne, elle peut donc avoir la valeur $\top$ (vrai) ou $\perp$ (faux). Lorsqu'une valeur est associé à une variable, celle-ci est dite *assignée*. Pour plus de facilité, nous définissons également un littéral assigné. Soit $a = \top$, alors son littéral positif (a) est considéré comme assigné à $\top$, tandis que son littéral négatif ($\neg a$) est assigné à $\perp$. Par contre, si $a = \perp$ alors son littéral positif (a) est considéré comme assigné à $\perp$, tandis que son littéral négatif ($\neg a$) est assigné à $\top$.

Soit la clause $l_1 \vee l_2 \vee \ldots \vee l_n$, celle-ci est dite satisfaite si au moins un de ses littéraux l_i est assigné à vrai. La formule en forme conjonctive $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ est dite satisfaite si chaque clause C_i est satisfaite. Si une formule est satisfaite, l'assignation la satisfiant est dite *modèle* de Σ et celle-ci est dite consistante. Si aucune assignation ne permet de satisfaire Σ, celle-ci est dite inconsistante ou insatisfaisable.

Le problème SAT est le problème consistant à déterminer si une formule booléenne sous forme CNF est consistante (satisfaisable) ou inconsistante (insatisfaisable).

Le problème SAT est un problème important en informatique car celui-ci est le premier problème à avoir été prouvé NP-complet [Cook, 1971]. Un problème NP-complet est un problème dont la solution peut être vérifié polynomialement par une machine de Turing [Turing, 1936] déterministe. De plus, pour tout problème NP-complet il existe une réduction polynomiale en un autre problème NP-complet.

Des avancées récente sur les algorithmes permettant de résoudre ce problème on permis son utilisation pratique dans de multiple cadres. Les programmes ayant pour but de résoudre ce problème sont appelés solveurs SAT.

7.2 Solveurs

Différents algorithmes ont été proposés au cours du temps pour résoudre le problème SAT. Parmi ceux-ci, il est intéressant de noter [Davis et al., 1962]. La procédure DLL est une recherche arborescente où les nœuds de l'arbre sont des variables et les deux branches partant de ce nœud sont les deux assignations possibles pour cette variable. Lorsque la recherche atteint un état inconsistant, un retour en arrière est effectué et l'algorithme essaye la prochaine branche non explorée. Ce retour en arrière consiste à annuler le dernier choix effectué ainsi que toute modification qui en résultait. Lorsqu'une variable est assignée, il est parfois possible de déduire la valeur pour d'autres variables. En effet, comme la procédure cherche un modèle, dès qu'une clause ne possède plus qu'un seul littéral non-affecté, ce dernier doit être assigné à $\top$ pour que la clause soit satisfaite. Ensuite, par effet de cascade, il est possible que la procédure puisse déduire la valeur d'autres littéraux. Ce procédé de déduction de valeur de littéraux est appelé propagation unitaire.

7.2.1 CDCL

Par la suite, différentes améliorations ont été proposées. Tout d'abord, une forme d'apprentissage fut proposée par [Marques-Silva and Sakallah, 1997]. Pour ce faire, un graphe d'implication est réalisé dans lequel les sommets représentent les littéraux assignés. À cela sont ajoutés des hyper-arcs représentant les clauses utilisées pour propager un littéral. Le littéral propagé à $\top$ est la destination de l'hyper-arc, tandis que les autres littéraux de la clause sont la source de cet hyper-arc. Lorsqu'un conflit apparait durant le cours de la recherche, deux hyper-arcs ont pour même destination la même variable, mais utilisant deux polarités différentes. À partir de ces deux clauses, l'une contenant l, l'autre $\neg l$, il est possible d'effectuer une résolution sur ces deux clauses. La clause obtenue est de type $l_1 \vee l_2 \vee \ldots \vee l_n$. Il est possible de continuer d'appliquer l'opérateur $\otimes_{\mathcal{R}}$ entre la résolvante obtenue précédemment et la raison des littéraux du même niveau que le littéral conflictuel jusqu'à obtenir un seul littéral du niveau courant.

On appelle généralement le dernier littéral du niveau courant ainsi obtenu le premier point implicant unique.

Lors de la propagation unitaire, l'algorithme de recherche doit retrouver les clauses étant devenues unitaires ou falsifiées. Pour se faire, il existe plusieurs structures de données. Initialement, il existait pour chaque littéral une liste de clause utilisant ce littéral, appelés listes d'occurences. Cependant, deux améliorations ont été proposées: les structures de données paresseuses [Zhang and Stickel, 2000] et les littéraux surveillés [Moskewicz et al., 2001].

En plus de ces structures de données, est apparut le principe de redémarrage. Après une certaine période donnée – exprimée la plupart du temps en nombre de conflits – l'algorithme annule l'ensemble des choix effectués. L'idée latente au concept de redémarrage est qu'ainsi, cela permet de diminuer l'impact de mauvaises premières décisions.

La dernière idée principale lié aux évolution d'un solveur DLL est la suppression d'un sous-ensemble des clauses apprises.

Lorsqu'un solveur DLL implémente ces différentes techniques: apprentissage, redémarrage, littéraux surveillés et oubli de clauses, on parle de solveur CDCL pour *conflict driven clause learning.*

7.2.2 Heuristiques

En plus de ces différents éléments, des heuristiques sont nécessaires.

Choix de variable

L'heuristique du choix de variable en cas de décision la plus connue et certainement la plus utilisé est celle du *Variable State Independent, Decaying Sum* (VSIDS). Celle-ci fut proposée par [Moskewicz et al., 2001] et améliorée par la suite. L'idée de base de cette heuristique est de valoriser les variables ayant été utilisées récemment. Un compteur d'activité est lié à chaque variable. Ensuite, lorsqu'une variable est utilisée dans la résolution de conflit, son activité est augmentée d'une valeur x. Régulièrement, cette valeur x augmente, permettant ainsi d'augmenter plus fortement les dernière variable actives et rattraper ainsi d'éventuels retards.

Suppression de clauses apprises

Le temps relatif à la propagation unitaire est directement lié au nombres de clauses que celle-ci doit envisager. De ce fait, garder un ensemble de clauses apprises le plus petit possible, tout en restant le plus utile possible est sujet à de nombreux travaux.

Parmi ceux-ci, on peut citer [Audemard and Simon, 2009a]. L'approche proposée est la suivante: soit C le nombre d'appel à la procédure permettant de réduire la base de clauses apprises. Alors, cette procédure aura lieu tout les $20000 + 500C$ conflits. En ce qui concerne le choix des clauses à supprimer, celui-ci se fait sur base de l'heuristique *lbd*. Celle ci fonctionne de la manière suivante: lorsqu'une clause est générée, l'algorithme calcule le nombre de niveaux de variables différents participent à cette clause. Ce niveau est appelé *lbd* d'une clause. Il a été observé expérimentalement que sur des instances dites *industrielles*, des clauses ayant un *lbd* équivalent à deux (appelées clause glues) étaient fortement utilisées dans la propagation. Lors d'un appel à la fonction relatif à la gestion des clauses apprises, seules les clauses ayant un *lbd* inférieur à la médianne des *lbd* des clauses apprises ou un *lbd* égal à 2 seront conservées.

Une autre méthode fut proposée dans [Audemard et al., 2011b]. À nouveau, soit C le nombre d'appel à la procédure permettant de réduire la base de clauses apprises alors la procédure se déroule tous les $500 + 100 \times C$. En ce qui concerne le choix des clauses, celui-ci se déroule en plusieurs étapes. Lorsqu'une réduction du nombre de clauses apprises est requis, la procédure évalue chaque clause. Pour chacune, la procédure calcule conceptuellement l'écart entre l'interprétation courante et cette dernière. Si la procédure voit que la clause ne devrait pas être falsifiée avant le prochain appel à la procédure, alors cette clause est retirée des structures de données nécessaires à la propagation. Cette clause est dite gelée. Cependant, si une clause gelée est suffisamment proche de l'interprétation courante, alors celle-ci est ré-activée. Lorsqu'une clause est gelée plus de sept fois de suite, alors celle-ci est considérée comme inutile et complètement supprimée de la mémoire.

Redémarrage

Différentes heuristiques permettant de savoir s'il faut redémarrer ont été proposées au fil des années. Récemment, l'heuristique *lbd* (pour *literal block distance*) [Audemard and Simon, 2009a] a obtenu un certain succès. L'heuristique de redémarrage à été conçue de manière à privilégier les clauses ayant une petite valeur de *lbd*. Pour ce faire, la moyenne glissante des n dernières valeurs de *lbd* est comparée à la moyenne de toutes les valeurs de *lbd*. Lorsque cette première devient trop importante vis-à-vis de cette seconde, un redémarrage est effectué.

7.3 Parallélisme

Depuis quelques années, l'horloge des différents processeurs accessibles sur le marché a stagné. Or, ces processeurs continuent d'être améliorés grâce notamment au parallélisme qui leur est apporté. Celui-ci peut se manifester par l'ajout d'unité arithmétique ou par des techniques de pipelining. Au delà de ce parallélisme qui apparait même sur nos téléphones portables, il est également possible de mettre en réseau différentes unités de calcul afin de distribuer des processus.

Cette augmentation de puissance de calcul en terme d'opération par seconde est regardée avec envie par presque toutes les communautés en informatique, y compris celle travaillant sur SAT. Pour utiliser cette puissance de calcul, il est nécessaire de changer ou d'augmenter l'algorithme séquentiel. Deux grande familles d'algorithmes existent en SAT parallèle: les méthodes portfolio et les méthodes diviser pour régner.

7.3.1 Portfolio

L'idée de base d'un portfolio est de réunir différents algorithmes traitant un même problème et de les lancer simultanément. Cela veut donc dire qu'un solveur portfolio utilisant n threads utilisera donc n solveur séquentiels. Un exemple d'un tel solveur est `ppfolio` [Roussel, 2012]. Celui-ci lance en parallèle de multiples solveur [Soos, 2010, Biere, 2013, Gebser et al., 2007, Wei and Li, 2009, Heule and van Maaren, 2009a] et les arrête tous dès qu'une réponse a été trouvée. Grâce à cette architecture, il est possible d'obtenir plus de résultat en un temps donné. Cependant, aucune réelle accélération ne peut être observée en doublant le nombre de threads.

Une deuxième approche du portfolio consiste à utiliser un même algorithme sur les différents threads, mais avec des constantes différentes pour les heuristiques. Et étant donné qu'un seul algorithme est exécuté simultanément, il lui est facile de partager différentes informations telles que les clauses apprises. Cependant, la quantité d'information échangée augmente fortement en fonction du nombre de threads. De ce fait, une politique de sélection des clauses à partager est primordiale pour obtenir de bons résultats. Un exemple d'un tel solveur est `ManySat`, proposé par [Hamadi et al., 2009b].

7.3.2 Diviser pour régner

La stratégie diviser pour régner réparti l'espace de recherche sur différents nœuds de calcul. Supposons que la formule initiale soit Σ. Si l'on fournit $\Sigma \wedge \Phi_i$ au $i^{\text{ième}}$ noeud de calcul, alors celui-ci verra son espace de recherche restreint par Φ_i. Toute la difficulté du principe de division pour régner dans le cadre de SAT est de déterminer une façon de générer ces Φ_i. En effet, cette formule permet de diviser l'espace de recherche, mais il est bien souvent impossible d'assurer l'aspect 'règne'.

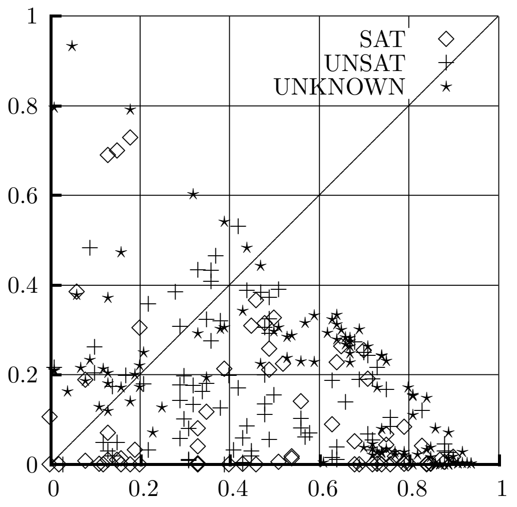
Différents heuristiques ont été proposées. L'une d'elle définit le concept de *guiding path* [Zhang et al., 1996], tandis que d'autres ont proposé une approche basée sur des conjonction de littéraux [Posypkin et al., 2012, Heule et al., 2012].

7.4 PeneLoPe: un solveur de type portfolio

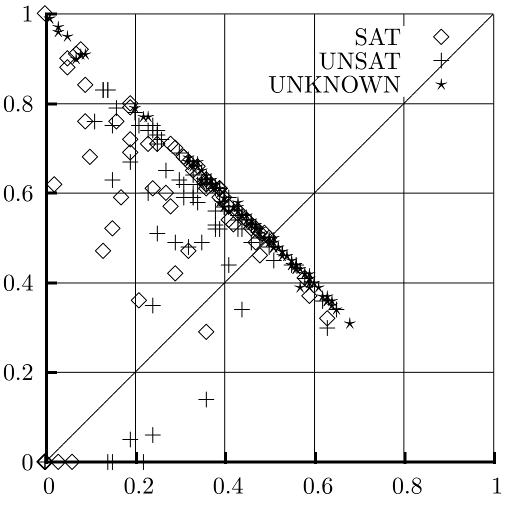
Cette première contribution à fait l'objet de publications [Audemard et al., 2012b, Audemard et al., 2012c] et de rapports techniques [Audemard et al., 2012a, Audemard et al., 2013a, Audemard et al., 2014a]. De plus, le solveur obtenu s'est classé deuxième durant le SAT Challenge 2012, troisième durant la SAT Competition 2013 et deuxième durant la SAT Competition 2014.

Dans un premier temps, nous avons conduit des expérimentations préliminaires afin d'étudier le comportement des solveurs *portfolio* vis-à-vis des clauses échangées. Pour un solveur séquentiel, une *bonne* clause apprise est une clause utilisée dans le processus de propagation unitaire et l'analyse de conflits. Pour les solveurs *portfolio*, on peut se baser naturellement sur la même idée : une *bonne* clause partagée est une clause qui aidera au moins un autre *thread* à réduire son espace de recherche, c'est à dire, qui participera à la propagation. Nous avons donc voulu connaître l'utilité des clauses partagées dans les solveurs *portfolio*.

Pour cela, nous avons mené des expérimentations réalisées sur des bi-processeurs `Intel XEON X5550` 4 cœurs à 2.66 GHz avec 8Mo de cache et 32Go de RAM, sous Linux CentOS 6 (kernel 2.6.32). Chaque solveur utilise 8 *threads*. Le temps limite alloué pour résoudre une instance est de 1200 secondes WC. Nous avons utilisé les 300 instances de la catégorie *application* de la compétition SAT 2011. Le solveur utilisé est `ManySat` 2.0 (basé sur `MiniSat` 2.2), l'un des solveurs *portfolio* les plus efficaces, et au sein duquel une vraie collaboration est réalisée. Ce solveur ne différencie les *threads* que sur les premières variables de décision, qui sont choisies au hasard. Ainsi, exceptée l'interprétation initiale, chaque *thread* associé à un solveur CDCL possède exactement le même comportement (en terme de redémarrages, heuristique de choix



(a) `ManySat`



(b) `ManySat` aggressif

Figure 7.1: Comparaison entre les clauses partagées utilisées et les clauses partagées non utilisées et supprimées. Chaque point correspond à une instance. L'axe des x donne le taux d'utilisation des clauses partagées $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, alors que l'axe des y donne celui des clauses partagées non utilisées et supprimées $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

de variable, etc.), ce qui permet de faire une comparaison plus juste, limitant les effets de bord liés aux autres paramètres. Par défaut, toutes les clauses apprises de taille inférieure à 8 sont partagées entre les *threads*. De plus, `ManySat` possède un mode déterministe [Hamadi et al., 2011] que nous avons utilisé afin de rendre reproductibles les différents résultats proposés dans cette contribution[1].

Soit $\mathcal{SC}$ l'ensemble des clauses partagées, c'est-à-dire l'union de toutes les clauses exportées par un *thread* vers les autres. Nous avons considéré deux types de clauses partagées. Tout d'abord, les clauses qui ont été utilisées (au moins une fois) par un *thread* dans le processus de propagation unitaire. Nous notons cet ensemble de clauses $used(\mathcal{SC})$. L'autre type de clauses considéré est celui des clauses qui ont été supprimées sans avoir du tout participé à la recherche. Nous le notons $unused(\mathcal{SC})$. Clairement, l'ensemble $\mathcal{SC}\backslash(used(\mathcal{SC}) \cup unused(\mathcal{SC}))$ représente les clauses qui, à la fois, n'ont pas encore été utilisées ni supprimées.

La Figure 7.1 illustre les résultats obtenus sur les différentes instances. Ils y sont reportés de la manière suivante : chaque point de la figure correspond à une instance, l'axe des x représentant le taux d'utilisation des clauses partagées ($\#used(\mathcal{SC})/\#\mathcal{SC}$) tandis que l'axe des y représente celui des clauses inutiles et supprimées ($\#unused(\mathcal{SC})/\#\mathcal{SC}$) et nous reportons la moyenne sur les différents *threads*. La Figure 7.1(a) donne les résultats pour `ManySat`. Nous pouvons tout d'abord remarquer que le taux d'utilisation des clauses partagées diffère fortement d'une instance à l'autre. Nous pouvons également remarquer que dans de nombreux cas, `ManySat` conserve des clauses pendant toute la recherche (points proches de l'axe des x). Ceci est dû à la politique non agressive du nettoyage des clauses apprises de `MiniSat` où dans de nombreux cas, très peu de nettoyages de clauses sont effectués. Les différents *threads* peuvent donc conserver les clauses inutiles durant une longue période, causant un sur-coût sans engendrer de bénéfice.

Nous avons réalisé les mêmes expérimentations en utilisant une politique de nettoyage des clauses beaucoup plus agressive, a savoir celle utilisée dans [Audemard et al., 2011b] (voir la section 4.3) et nous reportons les résultats dans la Figure 7.1(b). Ici, pour de nombreuses instances, les clauses sont supprimées avant d'avoir été utilisées et le taux de clauses utiles décroit fortement par rapport à la version basique de `ManySat`.

Ces résultats s'expliquent facilement. Si peu de nettoyages sont effectués, les différents solveurs doivent supporter de nombreuses clauses, qu'elles se révèlent utiles ou non. Cela fournit donc beaucoup d'informations et permet donc de propager plus de littéraux. Mais en contrepartie, les différents *threads* doivent alors maintenir les structures de données sur de nombreuses clauses inutiles, ralentissant ainsi le processus de recherche. Dans le cas contraire, si beaucoup de nettoyages sont réalisés, un autre problème survient. Si une clause partagée ne participe pas directement au processus de propagation et d'analyse de conflit, il y a de fortes chances qu'elle soit rapidement supprimée. Ainsi, les *threads* perdent beaucoup de temps à importer ces clauses, puis à les supprimer peu après.

Nous pouvons également noter que l'utilisation du *lbd* pour mesurer la qualité des clauses apprises ne semble pas non plus adéquate. En effet, les clauses partagées sont de petites clauses et ont donc une petite valeur de *lbd*. Les clauses importées ont alors plus de chance d'être préférées aux clauses générées par le *thread* lui-même. Ainsi, même s'il semble possible de

[1]Toutes les traces obtenues, ainsi que différentes statistiques, sont disponibles à `http://www.cril.fr/~hoessen/penelope.html`

paramétrer plus précisément la stratégie de nettoyage afin d'obtenir un solveur plus robuste, nous pensons que la gestion actuelle des clauses apprises n'est pas appropriée dans le cas du partage de clauses des solveurs *portfolio*. Nous proposons donc une nouvelle stratégie dans la prochaine partie.

7.4.1 Sélectionner, partager et activer les bonnes clauses

La gestion des clauses apprises est connue pour être un problème difficile dans le cas séquentiel. Gérer celles provenant d'autres fils d'exécution conduit nécessairement à de nouvelles difficultés :

- Les clauses importées peuvent être sous-sommées par des clauses déjà présentes. La sous-sommation étant une opération gourmande en temps, il est nécessaire d'offrir la possibilité de supprimer périodiquement certaines clauses apprises.
- Une clause importée peut être inutile durant une longue période avant d'être utilisée dans la propagation.
- Chaque fil d'exécution doit gérer un plus grand nombre de clauses.
- Il est très difficile de caractériser ce qu'est une bonne clause importée.

Pour chacune de ces raisons, nous proposons d'utiliser la politique de gestion dynamique des clauses apprises proposée par [Audemard et al., 2011b] au sein de chaque fil d'exécution. Cette technique récente permet d'activer ou de geler certaines clauses apprises, importées ou générées localement. L'avantage de cette technique est double. Étant donné qu'elles peuvent être gelées, la surcharge calculatoire occasionnée par les clauses importées est grandement réduite. De plus, les clauses importées, qui peuvent se révéler utiles dans un futur plus ou moins proche de la recherche, sont activées au moment adéquat. La prochaine section présente de manière plus précise cette méthode.

Geler certaines clauses

La stratégie proposée dans [Audemard et al., 2011b] est très différente de celles proposées par le passé (voir page 41). Elle est en effet basée sur le gel et l'activation dynamique des clauses apprises. À un certain point de la recherche, les clauses les plus prometteuses sont activées tandis que les autres sont gelées. De cette façon, les clauses apprises peuvent être écartées pour les prochaines propagations, mais peuvent par la suite être réactivées. Cette stratégie ne peut être utilisée avec les mesures d'activité basées sur le VSIDS ou le *lbd*. En effet, la mesure d'activité (inspirée par VSIDS) est dynamique mais ne peut être utilisée que pour mettre à jour l'activité des clauses actives -c'est-à-dire participant effectivement à la recherche- , tandis que *lbd* est soit statique (et ne change donc pas durant la recherche), soit dynamique auquel cas on rencontre le même problème que la mesure basée sur VSIDS. De ce fait, cette stratégie est associée avec une autre mesure afin de mesurer l'utilité d'une clause apprise [Audemard et al., 2011b].

Soient c une clause apprise par le solveur et ω l'interprétation courante résultant de la dernière polarité des variables de décisions [Pipatsrisawat and Darwiche, 2007]. La valeur *psm* de la clause c par rapport à ω, notée $psm_\omega(c)$, est égale à $psm_\omega(c) = |\omega \cap\ c|$.

Basée sur l'interprétation courante, *psm* se révèle être une mesure hautement dynamique. Son but est de sélectionner le contexte approprié à l'état courant de la recherche. Dans cette optique, les clauses ayant une petite valeur de *psm* sont considérées comme utiles. En effet, de telles clauses ont plus de chance de participer à la recherche, par le mécanisme de propagation unitaire ou en étant falsifiées. Au contraire, les clauses avec une grande valeur de *psm*, ont une plus grande probabilité d'être satisfaites par deux littéraux ou plus, les rendant inutiles pour la recherche en cours.

Ansi, seules les clauses ayant une petite valeur de *psm* sont sélectionnées et utilisées par le solveur dans le but d'éviter un surcoût calculatoire lié à la maintenance des structures de données pour ces clauses inutiles. Néanmoins, une clause gelée n'est pas supprimée, mais est gardée en mémoire, puisqu'elle peut se révéler utile par la suite. Au fur et à mesure que l'interprétation courante évolue, l'ensemble de clauses utilisées évolue également. Dans cette optique, la mesure *psm* est calculée périodiquement et les ensembles de clauses gelées et actives sont ainsi mis à jour.

Soit P_k une suite où $P_0 = 500$ et $P_{i+1} = P_i + 500 + 100 \times i$. Une procédure nommée "*updateDB*" est appelée chaque fois que le nombre de conflit atteint P_i (où $i \in [0..\infty]$. Cette procédure calcule la nouvelle valeur *psm* de chaque clause apprise (gelée ou active). Une clause avec une valeur *psm* inférieure à une limite l est activée, dans le cas contraire, elle est gelée. De plus, une clause qui n'est pas activée après k itérations (par défaut, $k = 7$) est définitivement supprimée. De façon similaire, une clause qui reste active plus de k étapes sans participer à la recherche est également supprimée.

Grâce aux mesures *psm* et *lbd*, il est désormais possible de définir une politique pour l'échange de clauses. Dans un solveur CDCL typique, un *nogood* est appris après chaque conflit. Il est donc clair que toutes les clauses ne peuvent être partagées. De ce fait, lorsqu'il y a collaboration, ces clauses doivent être filtrées selon un critère. À notre connaissance, dans tous les solveurs *portfolio*, ce critère est uniquement basé sur les informations de l'expéditeur de la clause, le destinataire n'ayant pas d'autres choix que d'accepter une clause jugée utile par un autre solveur.

Nous présentons donc une technique où l'expéditeur *et* le destinataire peuvent avoir leur propres stratégies. Chaque expéditeur (stratégie d'export) essaie de trouver dans sa base de clauses apprises les informations les plus pertinentes pour aider les autres *threads*. De plus, le destinataire (stratégie d'import) peut ne pas accepter aveuglément les clauses qui lui sont envoyées. Nous avons nommé ce solveur prototype `PeneLoPe`[2] (**P**arallel **L**bd **P**sm solver).

Politique d'import de clauses Quand une clause est importée, différents cas peuvent être considérés en fonction du moment où la clause sera attachée pour que celle-ci participe à la recherche.

- *no-freeze*: chaque clause importée est directement activée, et sera évaluée (et potentiellement gelée) durant le prochain appel à *updateDB* .
- *freeze-all*: chaque clause importée est *gelée* par défaut, et ne sera utilisée par la suite que si elle remplit les conditions pour être activée.

[2]En référence à la femme fidèle d'Ulysse, qui réalisait une tapisserie en liant de nombreux *fils* (*threads*)

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

Table 7.1: Comparaison entre les différentes stratégies d’import, d’export & redémarrage, en utilisant le mode déterministe

- *freeze*: chaque clause est évaluée au moment de l'import. Elle est activée si elle est considérée comme prometteuse, selon les mêmes critères que si elle avait été générée localement.

Politique d'export de clause Puisque `PeneLoPe` est capable de geler certaines clauses, il semble possible d'en importer un plus grand nombre que dans le cas d'une gestion classique des clauses, où chacune d'elle est attachée jusqu'à sa possible suppression. De ce fait, nous proposons différentes stratégies, plus ou moins restrictives, pour sélectionner les clauses partagées :

- *unlimited*: chaque clause générée est exportée aux différents fils d'exécution.
- *size limit*: seules les clauses dont la taille est inférieure à une valeur donnée (8 dans nos expérimentations) sont exportées [Hamadi et al., 2009b].
- *lbd limit*: une clause est exportée aux autres fils d'exécution si son *lbd* est inférieur à une limite donnée d (8 par défaut). Il est important de remarquer que la valeur du *lbd* peut être réduite au cours du temps. Ainsi, dès que $lbd(c)$ est inférieur à d, la clause est exportée.

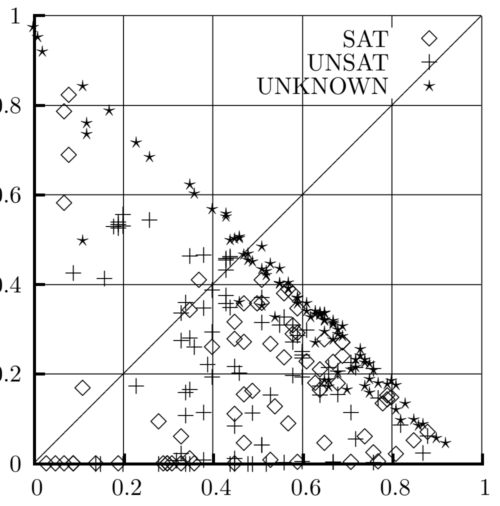
Politique de redémarrage En plus des politiques d'échange, `PeneLoPe` permet également de choisir entre deux politiques de redémarrage.

- *Luby*: Soit l_n le n-ième terme de la série Luby [Luby et al., 1993b]. Le n-ième redémarrage est effectué après $l_n \times \alpha$ conflits (α vaut 100 par défaut).
- *lbd* : Soit LBD_g la moyenne des valeurs de *lbd* de chaque clause apprise depuis le commencement de la recherche. Soit LBD_{100} la moyenne des 100 dernières clauses apprises. Cette politique induit qu'un redémarrage est effectué dès que $LBD_{100} \times \alpha > LBD_g$ (α vaut 0.7 par défaut) [Audemard and Simon, 2009a].

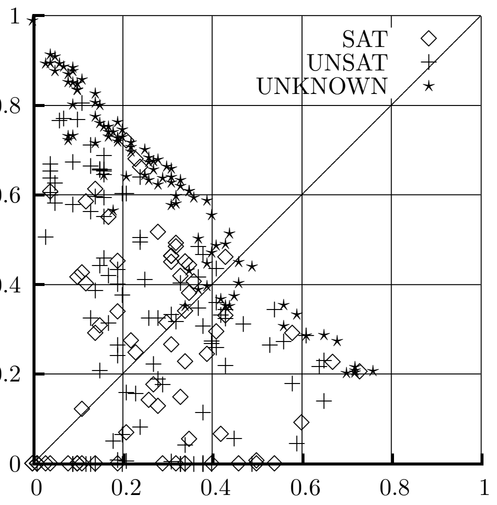
Différentes expérimentations ont été réalisées dans le but de comparer ces politiques d'import, d'export et de redémarrage. Différentes versions ont été exécutées et la table 7.1 présente une partie des résultats obtenus. Cette table rapporte pour chaque stratégie le nombre d'instances SAT (#SAT), UNSAT (#UNSAT) et total (#SAT + #UNSAT) résolues.

Comparons d'abord la stratégie d'import. Sans surprise, la politique *unlimited* obtient les plus mauvais résultats. En effet, aucune version utilisant cette stratégie n'est capable de résoudre plus de 190 instances. Chaque clause générée est exportée, et nous obtenons ainsi le niveau maximum de communication. Comme attendu, avec la multiplicité des fils d'exécution, les solveurs sont vite submergés par les clauses et leurs performances chutent.

C'est la raison pour laquelle des limites basées sur la taille ont été introduites avec l'idée que de petites clauses permettent de mieux filtrer l'espace de recherche et sont donc préférables. En effet, on constate dans la table 7.1 que la politique *size limit* surpasse *unlimited.* De plus, il est également possible de constater que l'échange des seules clauses unitaires n'offre pas d'aussi bonnes performances (*size = 1*), tout comme cela avait été annoncé préalablement. Il est également important de signaler que les clauses de grande taille peuvent grandement réduire la taille de la preuve [Beame et al., 2004].



(a) *size limit* + *luby* + *no freeze* (SLN)



(b) *lbd limit* + *lbd* + *freeze* (LLF)

Figure 7.2: Comparaison entre les clauses importées révélées utiles et les clauses importées n'ayant pas été utilisées dans la propagation et supprimées. Chaque point représente une instance. L'axe x représente le taux de clauses utiles $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, tandis que l'axe y représente le taux de clauses non utilisées et supprimées $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

L'utilisation de la valeur *lbd* d'une clause (*lbd*(c)) peut donc s'avérer bénéfique étant donné que *lbd*$(c) \leq$ taille(c). De ce fait, une politique utilisant *lbd* exporte plus de clauses que celle utilisant la taille (avec la même borne).

Ceci pourrait représenter un problème pour un solveur parallèle n'ayant pas la possibilité de geler certaines clauses. Cependant, puisque `PeneLoPe` contient un tel mécanisme, l'impact est grandement réduit. D'un point de vue empirique, la table 7.1 montre que *lbd limit* obtient les meilleurs résultats entre les différentes politiques.

Concentrons nous maintenant sur la politique de redémarrage. De façon évidente, la politique *luby* obtient globalement de moins bons résultats que celle utilisant *lbd*. Cela montre clairement l'intérêt de cette mesure introduite dans [Audemard and Simon, 2009a]. À propos de la stratégie d'import, aucune stratégie ne se révèle être clairement meilleure que les autres. En effet, le meilleur résultat relatif au nombre d'instances SAT est obtenu par *no freeze* (97) lorsqu'il est associée à la politique de redémarrage *luby* et *size limit* comme politique d'export, tandis que le meilleur résultat en nombre d'instance UNSAT est obtenu en utilisant la stratégie *freeze* (113). De plus, *no freeze* permet d'obtenir les meilleurs résultats globaux en résolvant 205 instances parmi les 300 utilisées. Il serait ainsi audacieux de plaider envers l'une des 3 techniques proposées. Cependant, un grand nombre des politiques proposées obtiennent de meilleurs résultats que les techniques "classiques" d'échange de clauses, représentées dans la table 7.1 par `ManySat`.

Dans une seconde expérimentation, nous avons voulu évaluer le comportement de `PeneLoPe` avec certaines des politiques proposées. Dans ce but, nous avons reconduit le même type d'expérimentations que celle présentée dans la section 4.2; les résultats obtenus sont reportés dans la Figure 7.2. Dans un premier temps, nous avons essayé avec les politiques *size limit*, *luby*, et *no freeze* (dénoté `SLN`, voir Figure 7.2(a)).

Il est clairement visible que cette version se comporte bien, étant donné que la plupart de ces points sont situés sous la diagonale. De plus, pour la plupart des instances, $\frac{\#usedSC+\#unused(SC)}{\#SC}$ est proche de 1 (nombreux points situés près de la seconde diagonale), ce qui indique que le solveur ne comporte que peu de clauses qui ne sont pas utilisées sans les avoir supprimées. La plupart d'entre elles sont donc utiles, tandis que les autres sont supprimées.

Ensuite, l'expérimentation fut reconduite en utilisant les politiques *lbd limit*, *lbd* , et *freeze* (dénoté `LLF`, voir Figure 7.2(b)). Au premier abord, le comportement de cette version est moins satisfaisant que la version `SLN`, du fait que pour la plupart des instances, plus de la moitié des clauses importées sont supprimées sans avoir été utiles dans la propagation. En réalité, dans cette version, un nombre bien plus important de clauses sont exportées du fait de la politique d'export *lbd limit*, ce qui conduit à un taux d'utilisation plus faible. Un réglage plus fin des paramètres (limite d'export pour les valeurs *lbd*, nombre de fois qu'une clause peut être gelée avant d'être supprimée, etc.) pourrait améliorer cela.

Si l'on regarde les statistiques détaillées de quelques instances, présentées dans la table 7.2, il apparaît que la version `LLF` partage en effet beaucoup plus de clauses que la version `SLN` (colonne nb_u). Notons aussi que cette table contient d'autres informations très intéressantes. Par exemple, il est possible de voir que pour certaines instances (par exemple `AProVE07-21`), quasiment 90% des clauses importées sont en fait gelées et ne participent pas immédiatement à la recherche, alors que pour d'autres instances la situation inverse se produit (`hwmcc...`). Ceci révèle la grande adaptabilité de la mesure *psm*.

Instance	Version	Temps	nb_c	nb_i	nb_f	nb_u	nb_d
dated-10-17-u	SLN	TO	1771	278 (0.15)	0%	45%	49%
	LLF	949	1047	1251 (0.83)	64%	20%	60%
hwmcc10-timeframe-	SLN	TO	5955	7989 (1.34)	0%	35%	60%
expansion-k50-eijkbs6669-tseitin	LLF	766	3360	15299 (4.55)	10%	11%	80%
velev-pipe-o-uns-1.1-6	SLN	150	981	69 (0.07)	0%	60%	24%
	LLF	48	296	173 (0.58)	41%	31%	33%
sokoban-sequential-p145-	SLN	TO	182	86 (0.47)	0%	92%	4%
microban-sequential.040	LLF	530	74	155 (2.09)	5%	58%	17%
AProVE07-21	SLN	10	78	83 (1.06)	0%	35%	16%
	LLF	31	143	506 (3.53)	89%	9%	57%
slp-synthesis-aes-bottom13	SLN	445	1628	194 (0.11)	0%	58%	30%
	LLF	91	309	298 (0.96)	71%	24%	49%
velev-vliw-uns-4.0-9-i1	SLN	TO	1664	262 (0.15)	0%	55%	40%
	LLF	906	1165	824 (0.70)	35%	37%	48%
x1mul.miter.shuffled-as.sat03-359	SLN	819	2073	421 (0.20)	0%	51%	37%
	LLF	280	680	1134 (1.66)	76%	16%	59%

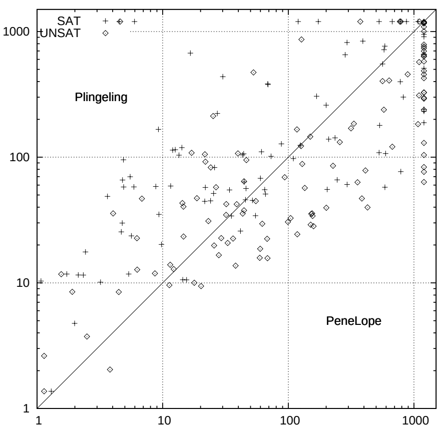
Table 7.2: Statistiques à propos d'instances UNSAT. Pour chacune d'entre elle et pour chaque version de `PeneLoPe`, nous reportons le temps WC nécessaire pour résoudre l'instance, le nombre de conflits (nb_c, en milliers), le nombre de clauses importées (nb_i, en milliers) avec entre parenthèses le taux entre nb_i et nb_c, le pourcentage de clauses gelée à l'import (nb_f), le pourcentage de clauses importées utiles (nb_u) et le pourcentage de clauses importées et supprimées sans avoir été utiles dans la recherche (nb_d). À l'exception du temps, chaque mesure est une moyenne sur les 8 fils d'exécution.

D'un point de vue plus général, même si la politique *no-freeze* semble être meilleure en terme d'efficacité des communications entre fils d'exécution, elle possède l'inconvénient d'ajouter chaque clause importée à l'ensemble des clauses actives. Ceci implique que le nombre de propagation par seconde sera ralenti jusqu'au prochain ré-examen de la base de clauses apprises. Ceci peut être un problème si l'on augmente le nombre de fils d'exécution. D'un autre coté, la politique *freeze-all* ne ralentit pas le solveur, mais dans ce cas, il est possible que le solveur soit en train d'explorer un espace de recherche qui aurait pu être évité avec la politique *no-freeze*.

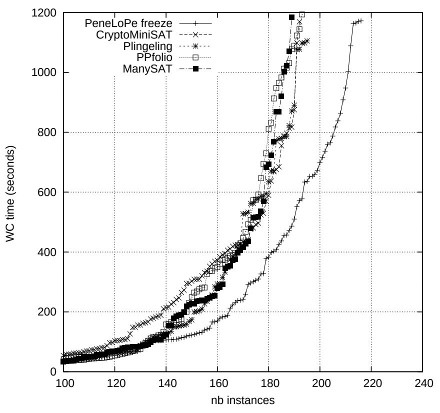
7.4.2 Evaluation

Solver	#SAT	#UNSAT	total
`PeneLoPe` *freeze*	97	**119**	**216**
`PeneLoPe` *no freeze*	96	**119**	215
`Plingeling` [Biere, 2013]	**99**	97	196
`ppfolio` [Roussel, 2011]	91	103	194
`cryptominisat` [Soos, 2010]	89	104	193
`ManySat` [Hamadi et al., 2009b]	95	92	187

(a) `PeneLoPe` VS l'état de l'art



(b) `PeneLoPe` *freeze* VS `Plingeling`



(c) Cactus plot

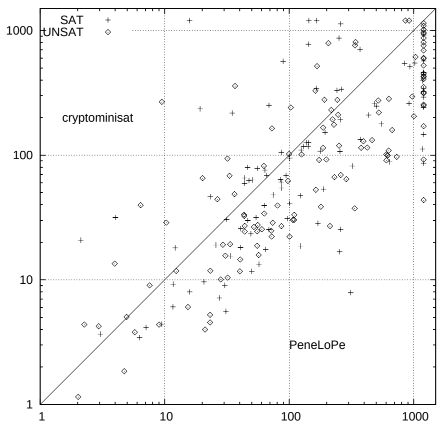
Figure 7.3: Comparaison sur 8 cœurs

Dans cette section, nous proposons une comparaison de deux de nos prototypes avec les meilleurs solveurs SAT parallèles connus à ce jour. Nous avons ainsi sélectionné les solveurs qui se sont montrés les plus efficaces lors des dernières compétitions : `ppfolio` [Roussel, 2011], `cryptominisat` [Soos, 2010], `Plingeling` [Biere, 2013] et `ManySat` [Hamadi et al., 2009b]. Pour `PeneLoPe`, nous avons choisi pour les deux versions la stratégie de redémarrage basée sur *lbd*, ainsi que la politique d'exportation *lbd limit*. Ces versions ne diffèrent donc que par leur politique d'importation de clauses, dont l'une est *freeze*, l'autre *no freeze*. Précisons également que contrairement à tous les résultats présentés précédemment, nous n'avons pas utilisé le mode déterministe dans cette partie, dans le but d'obtenir les meilleures performances possibles.

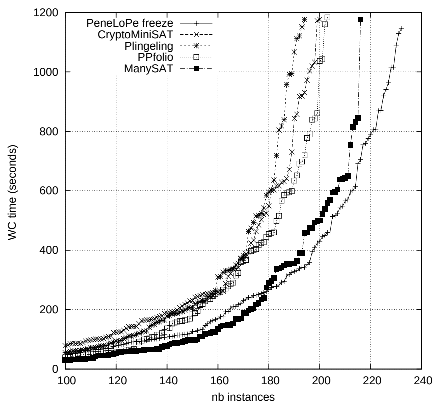
La Figure 7.3 donne les résultats selon différentes représentations. `PeneLoPe` dépasse en pratique tous les autres solveurs paralèlles. En effet, il parvient à résoudre 216 instances tandis qu'aucun autre solveur ne dépasse les 200 (Figure 7.3(a)). Notons cependant que si on ne considère que les instances satisfaisables, les meilleurs résultats sont obtenus par `Plingeling` qui en résout 99. Ceci est particulièrement visible dans la Figure 7.3(b) où `PeneLoPe` et `Plingeling` sont plus précisément comparés. Dans cette Figure, la plupart des points représentant des instances SAT sont assurément au dessus de la diagonale, illustrant la force de `Plingeling` sur ce type de problèmes. Toutefois, les résultats relatifs aux instances SAT sont assez proches les uns des autres (97 pour `PeneLoPe` *freeze*, 95 pour `ManySat`, etc.), l'écart étant plus important pour les problèmes UNSAT.

Solver	#SAT	#UNSAT	total
`PeneLoPe` *freeze*	104	127	**231**
`PeneLoPe` *no freeze*	99	**131**	230
`ManySat` [Hamadi et al., 2009b]	105	111	216
`ppfolio` [Roussel, 2011]	**107**	97	204
`cryptominisat` [Soos, 2010]	96	105	201
`Plingeling` [Biere, 2013]	100	95	195

(a) `PeneLoPe` VS l'état de l'art



(b) `PeneLoPe` *freeze* VS `cryptominisat`



(c) Cactus plot

Figure 7.4: Comparaison sur 32 cœurs

En outre, nous avons également comparé ces solveurs sur une architecture composée de 32 cœurs. Plus précisément, la configuration matérielle est maintenant la suivante : `Intel Xeon CPU X7550` (4 processeurs, 32 cœurs) 2.00GHz avec 18 Mo de mémoire cache et 256Go de mémoire vive.

Chaque solveur est exécuté avec 32 *threads*, et les résultats obtenus sont présentés dans la Figure 7.4 de manière similaire.

Tout d'abord, notons qu'à l'exception de `Plingeling`, tous les solveurs améliorent leurs performances quand ils sont exécutés avec un nombre supérieur de *threads*. Le profit est cependant limité pour certains d'entre eux. Par exemple, `cryptominisat` résout 193 instances avec 8 *threads*, et 201 instances avec 32 *threads*. L'amélioration est bien supérieure avec `PeneLoPe`, dont les deux versions résolvent 15 instances supplémentaires, et plus spectaculaire encore pour `ManySat` avec un gain de 29 instances. L'écart est tout particulièrement visible sur la Figure 7.4(c), puisque nos 3 compétiteurs résolvent le même nombre d'instances sur (environ) le même temps (les courbes de `ppfolio`, `Plingeling` et `cryptominisat` sont très proches les unes des autres), tandis que la courbe de PeneLoPe et celle de ManySat montrent clairement leur capacité à résoudre un plus grand nombre de problèmes en un temps restreint. Il est d'ailleurs bon de noter que PeneLoPe résout le même nombre de problèmes que Plingeling, ppfolio et cryptominisat avec une limite (virtuelle) de temps de seulement 400 secondes. Enfin, il semble que le comportement de PeneLoPe puisse être amélioré sur les instances SAT. En effet, il apparaît que les redémarrages *luby* sont plus efficaces pour les problèmes possédant une solution que pour ceux qui en sont dépourvus, alors que le contraire se produit pour le redémarrage *lbd*.

L'ajout d'unités de calcul, et par conséquent de fils d'exécution parallèles, a différents impacts. Par exemple, pour ppfolio et Plingeling, le gain n'est pas énorme, puisque l'augmentation du nombre de *threads* ne fait qu'accroître le nombre de solveurs séquentiels explorant l'espace de recherche ; chaque *thread* ne profite pas ici du travail des autres, puisqu'au sein de ces solveurs, aucune collaboration (ou une collaboration très faible) n'est effectuée. PeneLoPe bénéficie mieux de l'augmentation de ressources, car le nombre de clauses échangées provenant de différents sous-espaces de recherche est supérieur. Ceci conduit à une connaissance plus grande pour chaque *thread*, sans ralentir le processus de recherche général, grâce au mécanisme de gel des clauses.

Pour finir, insistons sur le fait que durant nos expérimentation avec PeneLoPe, tous les *threads* possèdent *exactement* les mêmes paramètres et les mêmes stratégies, comme dans nos expérimentations préliminaires présentées dans la section 4.2. Offrir de la diversification aux différentes recherches CDCL concurrentes devrait accroître plus encore l'efficacité pratique de notre prototype.

7.4.3 Conclusion

Dans cette contribution, nous avons proposé différentes stratégies permettant une meilleure gestion de l'échange de clauses apprises au sein de solveurs SAT parallèles de type *portfolio*. En se basant sur les concepts de *psm* et de *lbd* récemment proposés dans le cadre séquentiel, l'idée générale est d'adopter différentes stratégies pour l'importation et l'exportation de clauses. Nous avons étudié avec soin divers aspects empiriques des idées proposées, et comparé notre prototype aux meilleures implantations disponibles, montrant que notre solveur se révèle compétitif.

Assez clairement, diversifier le comportement exploratoire des *threads* devrait encore améliorer les performances de notre solveur, dans la mesure où cette diversification semble être la pierre angulaire de l'efficacité de certains solveurs comme ppfolio. Nous prévoyons d'étudier plus précisément cela dans un futur proche.

7.5 Dolius

Cette seconde contribution à été l'objet des publications suivantes: [Audemard et al., 2014c, Audemard et al., 2014e, Audemard et al., 2013b].

Le but de cette contribution est de fournir une plateforme capable de résoudre une instance du problème SAT en distribué. Pour cela, cette plateforme est destinée à diviser le travail par l'approche diviser pour régner. De plus, il est également possible de transmettre une partie des connaissances accumulées dans l'espoir de limiter au maximum le travail redondant.

7.5.1 Architecture

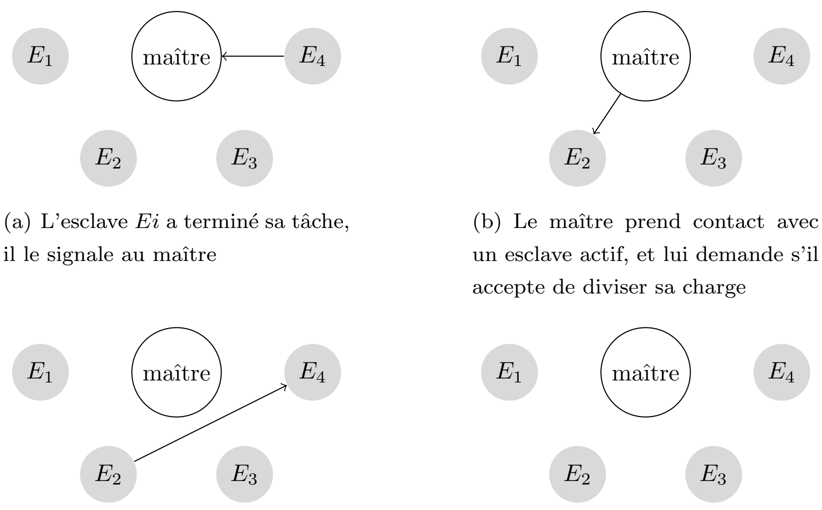
Structure générale / Initialisation

La structure générale de `dolius` est de type maître/esclave. Les esclaves sont des solveurs de type CDCL ; dans la pratique nous utilisons `PeneLoPe` [Audemard et al., 2012b], ce qui nous permet d'obtenir un solveur distribué dont les esclaves peuvent être des solveurs séquentiels "classiques", ou des portofolios de solveurs, permettant d'exploiter au mieux les architectures multi-cœurs; tandis que le maître est un processus dont la tâche est de mettre en relation les esclaves, quand l'un d'eux a terminé sa tâche (voir plus bas).

L'initiation de `dolius` consiste simplement à démarrer le maître. Celui-ci est alors en attente d'un ou plusieurs esclaves, qui entameront alors la résolution du problème. Il est à noter que l'implémentation `dolius` autorise l'ajout de ressources (esclaves) en cours de résolution. Ainsi, il est possible à n'importe quel moment de la recherche d'augmenter le nombre d'esclaves, ceux-ci n'ayant qu'à contacter le maître. Au travers de notre stratégie, une formule est prouvée satisfiable (SAT) si l'un des esclaves trouve un modèle. En effet, chaque esclave travaille avec la formule initiale simplifiée par la division du travail (voir section 4.3). Une formule est prouvée insatisfaisable (UNSAT) si tous les esclaves ont prouvé que la sous-formule dont ils s'occupent est UNSAT.

Rééquilibrage de charge

Il semble impossible en pratique de partitionner l'espace de recherche de manière à la fois statique et optimale au début de l'algorithme. Idéalement, les esclaves doivent avoir une charge de travail équivalente, mais il est en effet très difficile de prédire à l'avance la difficulté d'une (sous-)formule.



(a) L'esclave Ei a terminé sa tâche, il le signale au maître

(b) Le maître prend contact avec un esclave actif, et lui demande s'il accepte de diviser sa charge

(c) S'il accepte, l'esclave actif Ej (sélectionné par le maître) envoie une partie de sa charge à l'esclave actif Ei

(d) Tous les esclaves sont maintenant actifs

Figure 7.5: Illustration de l'équilibrage de charge via le vol de travail

Un mécanisme de rééquilibrage de charge de travail doit donc être mis en place, afin d'être opportuniste vis-à-vis des ressources disponibles. En effet, certains esclaves termine en pratique leur tâche bien avant certains autres, dans la large majorité des cas. Nous avons ainsi implanté au sein de notre plate-forme un mécanisme qui permet de répartir les tâches sur les différentes unités de calcul, ceci à l'aide d'un processus de plus en plus répandu : le *vol de travail*. L'avantage de cette technique est qu'un esclave cherchant une solution ne doit pas se soucier d'autre esclaves qui seraient affamés. Ceux-ci viendront d'eux même demander du travail. Les différentes étapes de ce processus sont illustrées en Figure 7.5. Lorsqu'un esclave (noté Ei) termine la tâche qui lui est confiée, il contacte le maître (M) pour signaler qu'il n'a pas plus de travail (étape 7.5(a)). Le maître demande à l'un des esclaves encore actifs (noté Ej) s'il accepte de diviser sa tâche (étape 7.5(b)). Si Ej accepte, il contacte alors Ei pour se délester d'une partie de sa charge de travail (étape 7.5(c)). Les esclaves entrent donc directement en communication l'un avec l'autre sans passer par le maître, qui n'est contacté que pour fournir à un esclave nouvellement inactif les coordonnées d'un autre esclave acceptant de diviser sa tâche.

Dans notre plate-forme, un esclave ne peut refuser de diviser son travail que lorsqu'un des trois cas suivants se présentent :

1. l'esclave vient de trouver un modèle à la formule, la recherche d'une solution devient donc inutile

2. il n'œuvre sur sa tâche que depuis peu de temps (en pratique ¡ x secondes, où x peut être spécifié par l'utilisateur. Par défaut, $x = 2, 5$)

3. il est déjà en cours de division de sa charge avec un autre esclave inactif

En outre, au sein de `dolius`, le maître conserve l'ensemble des esclaves actifs dans une file. Lorsqu'un esclave demande du travail, il sélectionne le premier esclave actif de sa file afin de le contacter. Ce choix a été fait pour éviter, en cas de demandes de travail simultanées par plusieurs esclaves inactifs, qu'un même esclave soit contacté pour plusieurs demandes de division de travail, ce qui serait très inefficace.

Dans la section suivante, nous détaillons les stratégies de division du travail offertes par `dolius`.

Division du travail

Lorsqu'il en reçoit la requête, la manière dont un esclave divise sa charge de travail pour en céder une partie est un facteur important de l'efficacité d'un algorithme de type diviser pour régner. La division sera considérée comme parfaite si les temps de résolution des sous-formules sont égaux entre eux, et inférieur au temps nécessaire pour résoudre la formule complète φ. Si la division est mal effectuée, deux problèmes de nature différente peuvent survenir. Premièrement, il est possible que les sous-formules φ_i aient un même temps de résolution que la formule complète. De ce fait, le temps nécessaire pour résoudre l'instance augmentera en fonction du nombre de ressources: le temps nécessaire pour résoudre l'instance auquel il faut ajouter le temps nécessaire pour la division du travail. Le deuxième problème lié à une mauvaise division se produit si le temps de résolution de φ_i est négligeable ou bien moindre que celui pour φ_j. Cela implique pour le système d'être capable de mettre en place une politique de ré-équilibrage de travail. Si cette politique souffre de mauvaise division, des cas pathologiques peuvent apparaitre, comme le cas dit *ping-pong* [Jurkowiak et al., 2001].

Exemple 1 *Soit ϕ une formule CNF. La formule $\Sigma = ((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ est également une CNF. Les divisions sur les variables a et b posent toutes les deux problème.*

(a) si la division du travail de Σ est réalisée sur la variable a, l'une des deux charges de calculs est alors très faible, car il est très facile de prouver que $\Sigma \vDash a$. Ainsi, par simple propagation unitaire, il est possible de montrer que $\Sigma \wedge (\neg a) \vDash \bot$. L'esclave ayant reçu cette partie du problème va donc la prouver incohérente sans la moindre exploration (la propagation unitaire étant suffisante), et redemander du travail très rapidement. Ceci est problèmatique, car la division du travail a un coût, notamment en transfert réseau. Le phénomène ping-pong est le fait de multiplier de tels mauvais choix pour la division du travail, menant à des sous-problèmes d'une trivialité telle que certains esclaves passent plus de temps à demander du travail qu'à véritablement participer à la résolution de la formule.

(b) si la division est réalisée sur b, alors chacun des deux esclaves travaille sur la même formule : $a \wedge \phi$. En outre, si ϕ est UNSAT, c'est seulement lorsque le moins efficace des deux (i.e. celui répondant en dernier) transmet sa solution que l'incohérence de Σ peut être établie.

Ainsi, dans une telle situation, il est largement préférable de diviser selon l'une des variables de ϕ plutôt que sur a ou b.

Dans l'idéal, la tâche doit être divisée en deux sous-tâches différentes de difficulté similaire, afin de répartir au mieux la charge de travail, tout en s'assurant que les sous-tâches soient de difficulté moindre que la tâche avant division. Malheureusement, comme précisé plus haut, il semble difficile sinon impossible de déterminer à l'avance la difficulté d'une formule. Les algorithmes diviser pour régner basent donc leur division sur des concepts heuristiques.

Les algorithmes de ce type utilisent généralement la notion de *guiding path* pour la division du travail. La plupart du temps le *guiding path* est réduit à la simple *séparation* d'une variable, et affecte (pour lui même) l'une des variables de la formule tandis qu'il transmet à l'esclave inactif l'opposé de cette même variable [Martins et al., 2010, Hyvärinen et al., 2011].

Notre plate-forme permet ce genre de division, mais se veut plus générique. En effet, `dolius` permet de diviser sa charge de travail selon une formule booléenne ϕ quelconque, l'un des esclaves considérant cette formule ϕ vraie, tandis que l'autre fait l'hypothèse que sa négation $\neg\phi$ est vraie. La division du travail a donc pour effet d'obtenir un arbre de résolution dont la racine est la formule initiale Σ et les feuilles les formules simplifiées par les *guiding paths* ϕ_i successifs. Un exemple d'un tel arbre est montré dans la Figure 7.6

Formellement, l'ensemble des *guiding paths* doit vérifier les conditions suivantes :

Propriété 1 *Soit Σ la formule propositionnelle à résoudre et $\phi_1, \phi_2, \ldots \phi_n$ les guidings paths successifs utilisés pour diviser Σ. La condition suivante doit alors être vérifiée : $\Sigma \wedge \bigwedge_{i=1}^{n} \phi_i$ est SAT.*

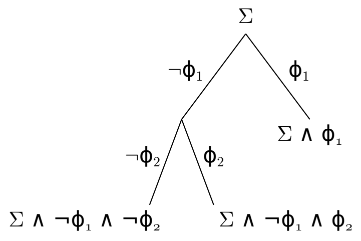

Σ
$\neg\phi_1$
ϕ_1
$\Sigma \wedge \phi_1$
$\neg\phi_2$
ϕ_2
$\Sigma \wedge \neg\phi_1 \wedge \neg\phi_2$
$\Sigma \wedge \neg\phi_1 \wedge \phi_2$

Figure 7.6: Un exemple de division de travail

Triviale dans le cas d'une division simple, cette propriété ne l'est plus dans le cas de divisions en utilisant des formules plus complexes. Nous étudions en ce moment des *guiding paths* dont la structure même nous assurent qu'ils sont mutuellement consistants. Clairement, qu'il s'agisse d'une simple variable ou d'une formule booléenne, le choix du *guiding path* est un élément déterminant pour l'efficacité de la procédure en général. Dans la première version de notre plate forme, nous avons fait le choix de conserver une division simple. Ainsi, l'esclave recevant une requête de division de travail sélectionne la variable ayant le VSIDS le plus important [Martins et al., 2010]. Cette valeur heuristique, utilisée au sein des solveurs CDCL comme choix de variable, est connue pour désigner les variables centrales de la formule [Simon and Katsirelos, 2012]. Ainsi, diviser sur la variable présentant le score le plus important a pour but de scinder la formule sur une variable jugée pertinente.

En pratique, chaque esclave est une instance de `PeneLoPe` travaillant sur plusieurs cœurs. Une difficulté supplémentaire apparait donc quand au choix de la variable de séparation. Pour l'instant, nous sélectionnons le cœur ayant généré le plus de conflits et sélectionnons sa "*meilleure*" variable. Il est clair que ce choix heuristique doit être amélioré et des expérimentations sont actuellement réalisées afin de déterminer un meilleur critère de sélection. Mais, à terme, nous souhaitons vivement déterminer une stratégie de séparation utilisant des formules plus complexes qu'une simple clause unitaire.

Transfert d'informations

Il est connu que la gestion des clauses apprises est un composant très important des solveurs CDCL [Audemard and Simon, 2009a] et des approches coopératives comme `ManySat` ou `PeneLoPe` [Hamadi et al., 2009b, Audemard et al., 2012b]. Dans le cas de notre approche distribuée ce problème perdure. Comme nous le montrerons dans la partie expérimentale, les performances de `dolius` s'écroulent si l'esclave en manque de travail ne reçoit pas de clauses apprises en plus du *guiding path*. En effet, l'esclave nouvellement créé doit alors recommencer la recherche depuis le début et ne profite d'aucun effort fait précédemment. Dès lors, il faut déterminer quelles clauses apprises les esclaves partagent. Dans cette première version de `dolius` + `PeneLoPe`, nous avons décider de partager toutes les clauses ayant un LBD [Audemard and Simon, 2009a] inférieur à 15. Les critères de sélection des nogoods à partager peuvent influer grandement, et des études plus approfondies sont encore nécessaires pour optimiser ce composant de `dolius`.

Un autre type d'information que nous n'avons malheureusement pas eu le temps de mettre en œuvre concerne l'initialisation de l'heuristique : il semble en effet intéressant d'initialiser la recherche dans le même état que l'esclave qui partage son travail. La structure de `dolius`

			5 esclaves			10 esclaves			20 esclaves		
Instance	SAT ?	`PeneLoPe`	min	moy	F	min	moy	F	min	moy	F
ACG-15-5p1	O	97	101	127	0	93	118	2	79	112	1
traffic_3_uc_sat	O	43	90	209	0	82	105	0	77	85	0
q_query_3_L70_coli.sat	O	74	81	2485	1	63	88	1	80	84	0
q_query_3_L150_coli.sat	N	150	288	368	0	319	440	0	345	534	0
traffic_pcb_unknown	N	269	239	301	0	194	279	0	176	221	0
SAT_dat.k80	N	770	-	-	5	-	-	5	-	-	5
sortnet-8-ipc5-h19-sat	O	752	1173	1173	4	720	720	4	-	-	5
aes_64_1_keyfind_1	O	-	568	568	4	128	372	1	716	718	3
partial-10-13-s	O	-	576	797	3	185	284	3	376	470	2
AProVE07-01	N	-	-	-	5	1101	1164	2	558	638	1
UTI-20-10p1	O	-	-	-	5	-	-	5	766	766	4
eq.atree.braun.12.unsat	N	-	-	-	5	-	-	5	354	354	4

Table 7.3: Résumé des résultats pour 12 instances. Pour chaque instance on affiche si elle est satisfiable, le temps de résolution nécessaire à `PeneLoPe`, et le temps minimum (min) moyen sur les runs réussis (moy) et le nombre d'echecs (F) pour 5, 10 et 20 esclaves.

```
//initialization
void setCNFFile(const char* inputFile);
void initialize(int nbVar, int nbClauses);
//thread related functions
void run();
void stop();
//clause database modification
void addLearntClause(const std::vector<int>& clause);
void addClause(const std::vector<int>& clause);
//iterators
void learntClauseIteratorRestart();
void learntClauseIteratorNext(std::vector<int>& clause);
void guidingPathIteratorRestart();
void guidingPathIteratorNext(std::vector<int>& clause);
int  getGuidingPathSize() const;
//guiding path modificators
bool createGuidingPath(std::vector<std::vector<int> >& gpA,
                       std::vector<std::vector<int> >& gpB);
void addToGuidingPath(const std::vector<int>& clauses);
//sat related information
bool solutionFound() const;
bool isSolutionFoundSAT() const;
int  getNbVar() const;
int  getSolutionLiteral(int var) const;
int  getNbLearntClauses() const;
```

Figure 7.7: Fonctions à implémenter pour s'intégrer à `dolius`

permet de telles approches, que nous prévoyons de tester dans de futurs travaux. C'est une des nombreuses perspectives offertes par `dolius`.

7.5.2 Interface de programmation

Afin de faciliter l'intégration de solveurs au sein de `dolius`, une interface de programmation est proposée à la Figure 7.7. Il existe différents groupes de fonctions: l'initialisation, la gestion des processus, la gestion des bases de clauses, les itérateurs, la modification des chemins de guidage et les informations relatives à la recherche.

7.5.3 Évaluation expérimentale

Les expérimentations conduites dans cette contribution sont réalisées sur des bi-processeurs `Intel XEON X5550` 4 cœurs à 2.66 GHz avec 8Mo de cache et 32 Go de RAM, sous Linux CentOS 6 (kernel 2.6.32). Chaque esclave utilise 8 *threads*. Le temps limite alloué pour résoudre une instance est de 1200 secondes WC (Wall Clock) (Les temps seront toujours donnés en secondes). Nous considérons ici le temps réel (WC) plutôt que le temps CPU car ce dernier n'intègre que le temps d'activité des unités de calcul, et occulte donc certains aspects, comme les temps d'attente des processus lors des communications réseau. Nous avons choisi un pool de 12 instances de difficulté variable issues de la compétition SAT 2011. Chaque instance a été testée avec 5, 10 et 20 esclaves.

Précisons qu'il est délicat d'évaluer expérimentalement une plate-forme de cette nature. C'est probablement l'une des raisons pour lesquelles depuis des années, aucun solveur de type diviser pour régner n'est soumis aux compétitions SAT. En effet, ce type d'approche n'est pas déterministe par nature, ni en temps d'exécution, ni dans la solution (preuve de réfutation ou modèle) reportée. Plusieurs exécutions de `dolius` + `PeneLoPe` sur un même ensemble de machines peuvent amener à des résultats disparates, dont la variance peut être plus ou moins grande. Ceci est inhérent à la plupart des processus distribués en informatique. Afin d'obtenir des résultats viables, nous avons donc lancé chaque instance à 5 reprises.

La table 7.3 donne une vue globale de l'ensemble des résultats obtenus. Nous avons également mis les résultats obtenus par `PeneLoPe` (correspondant à `dolius` avec un seul esclave).

					Attente		
E	Temps	MB	req	dernière	somme	moyenne	médiane
5	-	-	56	19.77	66.70	13.34	13.70
5	-	-	46	14.18	50.75	10.15	11.02
5	-	-	46	14.30	51.13	10.23	11.00
5	576	2	47	15.32	63.90	12.78	12.79
5	1019	1	48	14.10	50.23	10.05	10.92
10	-	-	141	27.30	143.47	14.35	14.93
10	-	-	160	32.97	162.61	16.26	16.79
10	-	-	211	31.40	201.61	20.16	20.86
10	383	4	164	42.26	175.99	17.60	18.17
10	185	2	182	24.37	161.81	16.18	16.49
20	-	-	472	35.15	376.62	18.83	19.16
20	-	-	466	37.34	352.25	17.61	18.80
20	637	8	415	30.82	344.25	17.21	17.21
20	376	7	438	49.17	330.61	16.53	16.95
20	398	5	479	21.05	325.88	16.29	17.58

Table 7.4: Détail des résultats pour l'instance partial-10-13-s. Chaque ligne correspond à un lancement. On y reporte le nombre d'esclaves (E), Le temps nécessaire à la résolution, le nombre de demandes de travail (req), la quantité totale transférée sur le réseau (MB), La dernière demande de travail par un esclave, la somme, moyenne, et médiane des temps d'attente de chaque esclave.

Jetons tout d'abord un œil aux 5 premières instances de ce tableau. Elles sont (relativement) faciles pour `PeneLoPe` et il semble que l'approche distribuée ne soit pas adaptée à ce type de formules simples. Pour de tels problèmes, notre implémentation fournit même de moins bons résultats qu'une approche centralisée, et dans certains cas, l'approche distribuée est d'ailleurs incapable de répondre à chaque exécution. Les 2 instances qui suivent (SAT_dat.k80 et sortnet-

8-ipc5-h19-sat) présentent plus de difficultés pour `PeneLoPe`. Ici, `dolius` obtient des résultats décevants, puisque dans les deux cas, l'approche contenant 20 esclaves ne parvient jamais à résoudre l'instance. Les versions contenant moins d'esclaves réussissent quant à eux une seule fois.

Les dernières instances listées dans la table 7.3 sont en revanche des instances difficiles, pour lesquels `PeneLoPe` ne réussit pas à déterminer la cohérence en moins de 1200 secondes. On commence à voir ici l'intérêt d'une approche distribuée puisque sur de telles instances, l'utilisation d'un certain nombre d'esclaves (chacun d'eux étant basés sur `PeneLoPe`) permet de trouver une solution là où l'approche portofolio seule montre ses limites. En effet, l'utilisation de ces ressources supplémentaires permet l'obtention d'une réponse lors de certaines exécutions (d'autres terminent en échec), en des temps de calcul très raisonnables pour des problèmes de cette difficulté.

Encore une fois, `dolius` est un projet de longue haleine, dont nous ne présentons ici qu'un travail préliminaire. Il est clair que `dolius` peut être plus finement paramétré afin d'accroître son efficacité pratique, toutefois les résultats obtenus par sa première version sont très prometteurs.

La table 7.4 donne des détails sur l'ensemble des runs de l'instance partial-10-13-s. Malheureusement, lorsque les instances ne sont pas résolues en 1200 secondes (-) nous n'avons pas la possibilité d'afficher les transferts réseaux réalisés. Comparons tout d'abord les lancements avec des nombres d'esclaves différents. Il est clair que le nombre de demandes de travail et la quantité transférée sur le réseau augmente avec le nombre de travailleurs. Il est par contre difficile de mettre en relation le temps d'exécution avec le nombre de travailleurs : nous ne sommes actuellement pas en mesure de proposer une méthode assurant une diminution du temps de calcul en augmentant le nombre de travailleurs.

La table 7.4 donne par contre des indications très intéressantes sur la division du travail. En effet, pour un nombre de travailleurs donné, le nombre de requêtes demandées et le temps où tous les esclaves travaillent jusqu'à la fin sont relativement proches y compris lorsque le solveur échoue à trouver une solution après 1200 secondes. Dans ce cas, la division de travail échoue quelque peu : Aucun espace de recherche n'est prouvé insatisfiable et l'affectation de certaines variables n'aide pas à la détection d'un modèle ; d'où l'importance du choix du *guiding path*.

7.5.4 Conclusion

Dans cette contribution, nous avons proposé un cadre permettant de résoudre le problème SAT de manière parallèle. Cette première version est capable de résoudre des instances très difficiles pour les solveurs centralisés. Il est de toute façon clair qu'une approche distribuée n'est utile que pour des instances très difficiles, le surcoût dû aux différents demandes de travail, transfert, etc. s'avérant rédhibitoires sur des instances dont le temps de résolution séquentiel est relativement court.

Clairement, le choix du *guiding path* est un élément important pour un tel algorithme. La séparation sur une seule variable (celle avec le VSIDS le plus élevé) permet d'obtenir des premiers résultats relativement satisfaisants. Il est toutefois clair qu'il est possible d'améliorer l'efficacité de la procédure en divisant avec soin le travail à accomplir. L'utilisation de formules plus générales que de simples clauses unitaires constitue également un point qu'il est important d'étudier.

De manière plus générale, ce travail offre de nombreuses perspectives de travail. Le choix de l'information à transférer entre esclaves, le temps avant qu'un esclave accepte de partager son travail, l'étude des cas pathologiques comme le cas du *ping-pong* sont autant de pistes de recherches que nous allons étudier dans les prochains mois.

Enfin, en mettant en place une API qui permettra à tout un chacun d'utiliser la plateforme `dolius`, nous espérons contribuer à l'étude des solveurs SAT sur les environnements distribués.

7.6 Perspectives

Au sein de cette thèse, il a été montré différents moyens de résoudre le problème SAT au travers de la programmation parallèle et distribuée. Différents points peuvent être retenus de cette expérience. En premier lieu, nous avons pu remarquer que des techniques qui semblent simple comme le portfolio sont capable de battre des méthodes plus complexes. Cependant, ces techniques ne sont pas aussi "stupides" qu'il apparait en premier lieu. Leur succès vient du principe "there is no free launch" et joue fortement en leur faveur et selon la méthode d'évaluation proposée par les compétitions. De plus, les portfolio sont capable de profiter directement des avancées scientifiques obtenues sur les solveurs séquentiels.

Deuxièmement, nous avons proposé deux méthodes capable de s'étendre en fonction des ressources proposées. La première, un portfolio est basé sur des travaux au sein de différents laboratoires et prennent pleinement avantage des mémoires partagées. La seconde, une plateforme permettant d'utiliser le concept de diviser pour régner. Cette nouvelle plateforme réduit le cout d'entrée pour les chercheurs souhaitant distribuer leurs calculs en étant moins dépendant du matériel sous-jacent. De plus, la grande flexibilité offerte pour la méthodologie des divisions pourra être d'une grande utilité pour de futures recherches.

La thèse n'est qu'un point de départ pour une carrière de recherche scientifique. Une fois ce premier pas établi, il est nécessaire de prendre du recul pour se lancer vers les prochains pas. Différentes possibilités sont offertes après le travail fourni dans cette thèse. En ce qui concerne la recherche utilisant le principe de division pour régner, de nombreuses heuristiques sont manquantes que ce soit concernant la division du travail, les communications. D'autre part, il est également envisageable de tester à intégrer des méthodes qui ont fait leur succès pour les solveurs séquentiels, tel que le redémarrage sans perte d'informations. Finalement, l'essentiel des efforts de cette thèse se sont porté sur la partie recherche de solution. Cependant, un solveur SAT est utilisé la plupart du temps dans une 'chaine' avec entre autres, d'éventuels préprocesseurs. Il serait intéressant de scruter chacun des élément de cette chaine pour voir s'il serait judicieux d'utiliser le parallélisme sur ceux-ci.

Les différentes techniques considérées durant cette thèse utilisaient la résolution pour résoudre les instances. Cependant, d'autres techniques devraient être considérées car celles-ci pourraient se révéler plus judicieuses dans le cadre de programmation parallèle. Il est envisageable que de telles techniques soient plus efficaces que celles présentées dans cet ouvrage. Cependant, de nombreux obstacles devront être relevés pour que celles-ci soient compétitives. En effet, les solveurs CDCL modernes résultent de plusieurs années d'optimisation, certains allant jusqu'à jouer avec des bits d'adresses mémoire pour économiser de l'espace et donc améliorer l'utilisation des caches CPU. De ce fait, de nouvelles méthodes devraient être suffisamment puissantes pour surpasser les solveurs SAT actuels.

CHAPTER 8

Contributions

There is no substitute for hard work.
—Thomas A. Edison

8.1 International peer-reviewed

- Gilles Audemard, Benoît Hoessen, Said Jabbour, Cédric Piette, "**Dolius: A Distributed Parallel SAT Solving Framework**", in *Pragmatics of SAT 2014 (POS'2014)*, Vienna, july 2014. [Audemard et al., 2014c]
- Gilles Audemard, Benoît Hoessen, Said Jabbour, Cédric Piette, "**An Effective Distributed D&C Approach for the Satisfiability Problem**", in *22nd International Conference on Parallel, Distributed and Network-Based Computing (PDP'14)*, pp 183–187, Turin, february 2014. [Audemard et al., 2014e]
- Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette, "**Revisiting Clause Exchange in Parallel SAT Solving**", in *Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, june 2012. [Audemard et al., 2012b]

8.2 National peer-reviewed

- Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette, "**Résolution parallèle de SAT : mieux collaborer pour aller plus loin**", in *8èmes Journées Francophones de Programmation par Contraintes (JFPC'12)*, may 2012 [Audemard et al., 2012c]
- Gilles Audemard, Benoît Hoessen, Said Jabbour, Cédric Piette, "**Un nouveau cadre diviser pour régner pour SAT distribué**", in *9èmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, june 2013. [Audemard et al., 2013b]

8.3 Technical reports

- Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette, "**PeneLoPe, a parallel clause-freezer solver**", in *Proceedings of SAT Challenge 2012: Solver and Benchmarks Descriptions* , pp. 43–44, may 2012 [Audemard et al., 2012a]
- Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette, "**PeneLoPe in SAT Competition 2013**", in *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Helsinki, may 2013. [Audemard et al., 2013a]
- Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette, "**PeneLoPe in SAT Competition 2014**", in *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, Helsinki, may 2014. [Audemard et al., 2014a]

8.4 Softwares

- `PeneLoPe`, a parallel SAT solver [Audemard et al., 2014b]
- `dolius`, a framework to distribute SAT solvers [Audemard et al., 2014d]
- a `MiniSat` version implementing the API for the Dolius platform [Audemard et al., 2014f]
- `saturnin`, a SAT solver developped with educationnal purpose [Hoessen, 2014]

List of Symbols

#	The cardinality operator
$\overline{\wedge}$	The nand operator
$\perp$	The minimal unsatisfiable core
$\equiv$	The equivalence operator
$\wedge$	The conjunction operator
$\vee$	The disjunction operator
$\mathcal{H}(\overline{x}, \overline{y})$	Hamming distance between two vectors
$\mathcal{O}$	Big-O notation
$\models$	is logical consequence
$\otimes$	The exclusive or operator
$\otimes_{\mathcal{R}}$	The resolution operator
$\Pi(\Sigma)$	the set of variables occurring in Σ
$\Pi^+(\Sigma)$	the set of variables occurring positively in Σ
$\Pi^-(\Sigma)$	the set of variables occurring negatively in Σ
$\Rightarrow$	The implication operator
$\top$	The minimal tautology
$\underline{\vee}$	The nor operator
$\Sigma\|_l$	The formula Σ simplified by assigning the literal l to $\top$

Index

Bibliography

[Amdahl, 1967] Gene M Amdahl (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM.

[Ansótegui et al., 2012] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy (2012). The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 410–423. Springer.

[Audemard et al., 2008] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs (2008). A generalized framework for conflict analysis. In *11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science (LNCS)*, pages 21–27. Springer.

[Audemard et al., 2012a] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2012a). Penelope, a parallel clause-freezer solver. *Proceedings of SAT Challenge 2012: Solver and Benchmarks Descriptions*, pages 43–44.

[Audemard et al., 2012b] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2012b). Revisiting clause exchange in parallel SAT solving. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 200–213. Springer.

[Audemard et al., 2012c] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2012c). Résolution parallèle de SAT : mieux collaborer pour aller plus loin. In *8ièmes Journées Francophones de Programmation par Contraintes (JFPC'12)*, pages 35–44, Toulouse.

[Audemard et al., 2013a] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2013a). Penelope in SAT competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmarks Descriptions*, page 66.

[Audemard et al., 2014a] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2014a). Penelope in SAT competition 2014. *Proceedings of SAT Competition 2014; Solver and Benchmarks Descriptions*, pages 58–59.

[Audemard et al., 2014b] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette (2014b). Penelope sources. `https://bitbucket.org/bhoessen/penelope`. Online; accessed 04-September-2014.

[Audemard et al., 2013b] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette (2013b). Un nouveau cadre diviser pour régner pour SAT distribué. In *9èmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 51–58.

[Audemard et al., 2014c] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette (2014c). Dolius: A distributed parallel SAT solving framework.

[Audemard et al., 2014d] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette (2014d). Dolius sources. `https://bitbucket.org/bhoessen/dolius`. Online; accessed 04-September-2014.

[Audemard et al., 2014e] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette (2014e). An effective distributed D&C approach for the satisfiability problem. In *22nd Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'14)*.

[Audemard et al., 2014f] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette (2014f). Minisat sources modified to be used with dolius. `https://bitbucket.org/bhoessen/dolius`. Online; accessed 04-September-2014.

[Audemard et al., 2011a] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs (2011a). On freezeing and reactivating learnt clauses. In *Fourteenth International Conference on Theory and Applications of Satisfiability Testing(SAT'11)*, pages 188–200. Best paper award.

[Audemard et al., 2011b] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs (2011b). On freezeing and reactivating learnt clauses. In *proceedings of SAT*, pages 147–160.

[Audemard and Simon, 2009a] Gilles Audemard and Laurent Simon (2009a). Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404.

[Audemard and Simon, 2009b] Gilles Audemard and Laurent Simon (2009b). Predicting learnt clauses quality in modern sat solvers. In Craig Boutilier, editor, *IJCAI*, pages 399–404.

[Audemard and Simon, 2012] Gilles Audemard and Laurent Simon (2012). Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming*, pages 118–126. Springer.

[Audemard and Simon, 2014] Gilles Audemard and Laurent Simon (2014). Lazy clause exchange policy for parallel SAT solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 197–205. Springer International Publishing.

[Balint et al., 2013] Adrian Balint, Marijn JH Heule, Anton Belov, and Matti Järvisalo (2013). The application and the hard combinatorial benchmarks in SAT competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 99.

[Bayardo and Schrag, 1997] Roberto J Bayardo and Robert Schrag (1997). Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208.

[Beame et al., 2004] Paul Beame, Henry Kautz, and Ashish Sabharwal (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351.

[Bellman, 1978] R. Bellman (1978). *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser.

[Biere, 2008] Armin Biere (2008). Picosat essentials. *JSAT*, 4(2-4):75–97.

[Biere, 2013] Armin Biere (2013). Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 51. `http://fmv.jku.at/lingeling`.

[Biere, 2014] Armin Biere (2014). Lingeling essentials, a tutorial on design and implementation aspects of the the SAT solver lingeling. In Daniel Le Berre, editor, *POS-14*, volume 27 of *EPiC Series*, pages 88–88. EasyChair.

[Charniak and McDermott, 1985] E.A. Charniak and D.V. McDermott (1985). *Introduction to Artificial Intelligence*. Addison-Wesley Series in Computer Science Series. Addison-Wesley Pub.Company.

[Cook, 1971] Stephen A. Cook (1971). 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. ACM.

[Cook and Reckhow, 1979] Stephen A Cook and Robert A Reckhow (1979). The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(01):36–50.

[Darwiche, 2001] Adnan Darwiche (2001). Decomposable negation normal form. *J. ACM*, 48(4):608–647.

[Davis et al., 1962] Martin Davis, George Logemann, and Donald Loveland (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215.

[Dequen et al., 2009] Gilles Dequen, Pascal Vander-Swalmen, and Michaël Krajecki (2009). Toward easy parallel sat solving. In *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 425–432, Washington, DC, USA. IEEE Computer Society.

[Dubois and Dequen, 2001] Olivier Dubois and Gilles Dequen (2001). A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *IJCAI*, volume 1, pages 248–253.

[Encyclopædia Britannica, 2013] Encyclopædia Britannica (2013). propositional calculus.

[Eén and Biere, 2005] Niklas Eén and Armin Biere (2005). Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75. Springer.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson (2003). An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer.

[Eén and Sörensson, 2004] Niklas Eén and Niklas Sörensson (2004). An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer.

[Freeman, 1995] Jon William Freeman (1995). *Improvements to propositional satisfiability search algorithms*. PhD thesis, Citeseer.

[Gabriel et al., 2004] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Gebser et al., 2007] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub (2007). clasp: A conflict-driven answer set solver. In *Logic Programming and Non-monotonic Reasoning*, pages 260–265. Springer.

[Glover, 1989] Fred Glover (1989). Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206.

[Goldberg, 2006] Eugene Goldberg (2006). Determinization of resolution by an algorithm operating on complete assignments. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 90–95. Springer.

[Goldberg, 2008] Eugene Goldberg (2008). A decision-making procedure for resolution-based SAT-solvers. In *Theory and Applications of Satisfiability Testing–SAT 2008*, pages 119–132. Springer.

[Goldberg and Novikov, 2007] Eugene Goldberg and Yakov Novikov (2007). Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561.

[Gomes et al., 2000] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100.

[Gomes et al., 1998] Carla P Gomes, Bart Selman, Henry Kautz, et al. (1998). Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437.

[Gulati and Khatri, 2010] Kanupriya Gulati and Sunil P Khatri (2010). Boolean satisfiability on a graphics processor. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 123–126. ACM.

[Hamadi et al., 2011] Youssef Hamadi, Saïd Jabbour, Cédric Piette, and Lakhdar Saïs (2011). Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132.

[Hamadi et al., 2009a] Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs (2009a). Control-based clause sharing in parallel SAT solving. In *proceedings of IJCAI*, pages 499–504.

[Hamadi et al., 2009b] Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs (2009b). Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262.

[Haralick and Elliott, 1980] Robert M Haralick and Gordon L Elliott (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313.

[Haugeland, 1985] John Haugeland (1985). *Artificial intelligence: the very idea.* Massachusetts Institute of Technology, Cambridge, MA, USA.

[Hentenryck and Michel, 2009] Pascal Van Hentenryck and Laurent Michel (2009). *Constraint-based local search.* The MIT Press.

[Heule and van Maaren, 2009a] Marijn Heule and Hans van Maaren (2009a). march_hi. *SAT 2009 competitive events booklet: preliminary version*, page 27.

[Heule et al., 2012] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere (2012). Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing*, pages 50–65. Springer.

[Heule and van Maaren, 2009b] Marijn JH Heule and Hans van Maaren (2009b). Look-ahead based SAT solvers. *Handbook of Satisfiability*, 185:155–184.

[Hoessen, 2014] Benoît Hoessen (2014). Saturnin sources. `https://bitbucket.org/bhoessen/libsaturnin`. Online; accessed 04-September-2014.

[Hyvärinen et al., 2006] Antti Eero Johannes Hyvärinen, Tommi Junttila, and Ilkka Niemelä (2006). A distribution method for solving SAT in grids. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 430–435. Springer.

[Hyvärinen et al., 2011] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä (2011). Grid-based SAT solving with iterative partitioning and clause learning. In *proceedings of CP*, pages 385–399.

[Irfan et al., 2013] Ahmed Irfan, Davide Lanti, and Norbert Manthey (2013). Pcasso–a parallel cooperative SAT solver. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 64.

[Jabbour et al., 2012] Saïd Jabbour, Nadjib Lazaar, Youssef Hamadi, and Michèle Sebag (2012). Cooperation control in Parallel SAT Solving: a Multi-armed Bandit Approach. In *Workshop on Bayesian Optimization & Decision Making*, Lake Tahoe, United States.

[Jurkowiak et al., 2001] Bernard Jurkowiak, Chu Min Li, and Gil Utard (2001). Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189.

[Järvisalo et al., 2010] Matti Järvisalo, Armin Biere, and Marijn Heule (2010). Blocked clause elimination. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 129–144. Springer.

[Katsirelos et al., 2013] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon (2013). Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. *AAAI'13.*

[Kautz et al., 2009] Henry A Kautz, Ashish Sabharwal, and Bart Selman (2009). Incomplete algorithms. *Handbook of Satisfiability*, 185:185–204.

[Kottler and Kaufmann, 2011] Stephan Kottler and Michael Kaufmann (2011). SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*.

[Kurzweil, 1990] Raymond Kurzweil (1990). *The Age of Intelligent Machines*. Massachusetts Institute of Technology, Cambridge, MA, USA.

[Le Berre, 2001] Daniel Le Berre (2001). Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80.

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

[Li, 1999] Chu Min Li (1999). A constraint-based approach to narrow search trees for satisfiability. *Information processing letters*, 71(2):75–80.

[Luby et al., 1993a] Michael Luby, Alistair Sinclair, and David Zuckerman (1993a). Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180.

[Luby et al., 1993b] Michael Luby, Alistair Sinclair, and David Zuckerman (1993b). Optimal speedup of Las Vegas algorithms. In *proceedings of ISTCS*, pages 128–133.

[Mano and Kime, 2008] M. Morris Mano and Charles R. Kime (2008). *Logic and Computer Design Fundamentals (4th ed.)*. Pearson Education, Inc., Upper Saddle River, NJ, USA.

[Marques-Silva and Sakallah, 1997] Joao P. Marques-Silva and Karem A Sakallah (1997). Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society.

[Martins et al., 2010] Ruben Martins, Vasco Manquinho, and Ines Lynce (2010). Improving search space splitting for parallel SAT solving. In *Proceedings of ICTAI'10*, pages 336–343.

[Moore, 1965] Gordon Earle Moore (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Moskewicz et al., 2001] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

[Nadel and Ryvchin, 2012] Alexander Nadel and Vadim Ryvchin (2012). Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 242–255. Springer.

[Nilsson, 1998] N.J. Nilsson (1998). *Artificial Intelligence: A New Synthesis*. The Morgan Kaufmann Series in Artificial Intelligence Series. MORGAN KAUFMAN PUBL Incorporated.

[Nvidia, 2007] Nvidia (2007). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA Corporation.

[Piette et al., 2008] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs (2008). Vivifying propositional clausal formulae. In *ECAI*, volume 178, pages 525–529.

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

[Poole et al., 1998] D. Poole, A. Mackworth, and R. Goebel (1998). *Computational Intelligence: a logical approach.* Oxford University Press, Oxford.

[Posypkin et al., 2012] Mikhail Posypkin, Alexander Semenov, and Oleg Zaikin (2012). Using boinc desktop grid to solve large scale SAT problems. *Computer Science*, 13(1):25–34.

[RFC, 1981a] RFC (1981a). *RFC 791 Internet Protocol - DARPA Internet Programm, Protocol Specification.* Internet Engineering Task Force.

[RFC, 1981b] RFC (1981b). *RFC 793 Transmission Control Protocol - DARPA Internet Programm, Protocol Specification.* Internet Engineering Task Force.

[Rich and Knight, 1991] E. Rich and K. Knight (1991). *Artificial intelligence.* Artificial Intelligence Series. McGraw-Hill.

[Roussel, 2011] Olivier Roussel (2011). ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio`.

[Roussel, 2012] Olivier Roussel (2012). Description of ppfolio 2012. *Proceedings of SAT Challenge 2012: Solver and Benchmarks Descriptions*, page 46.

[Russell and Norvig, 2003] Stuart J. Russell and Peter Norvig (2003). *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition.

[Régin et al., 2013] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert (2013). Embarrassingly parallel search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer.

[Semenov and Zaikin, 2013] Alexander Semenov and Oleg Zaikin (2013). On estimating total time to solve SAT in distributed computing environments: Application to the SAT@home project. *CoRR*, abs/1308.0761.

[Siegel, 1987] Pierre Siegel (1987). Représentation et utilisation de la connaissance en calcul propositionnel. Thèse d'État. Luminy, Marseille.

[Simon, 2014] Laurent Simon (2014). Post mortem analysis of sat solver proofs. In Daniel Le Berre, editor, *POS-14*, volume 27 of *EPiC Series*, pages 26–40. EasyChair.

[Simon and Katsirelos, 2012] Laurent Simon and George Katsirelos (2012). Eigenvector centrality in industrial SAT instances. In *Proceedings of CP'12*, pages 348–356.

[Sonobe et al., 2014] Tomohiro Sonobe, Shuya Kondoh, and Mary Inaba (2014). Community branching for parallel portfolio SAT solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 188–196. Springer International Publishing.

[Soos, 2010] Mate Soos (2010). Cryptominisat 2.5. 0. *SAT Race competitive event booklet.* `http://www.msoos.org/cryptominisat2/`.

[Stone et al., 2010] John E. Stone, David Gohara, and Guochun Shi (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

[Sörensson and Biere, 2009] Niklas Sörensson and Armin Biere (2009). Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 237–243. Springer.

[Sörensson and Eén, 2005] Niklas Sörensson and Niklas Eén (2005). Minisat v1. 13-a SAT solver with conflict-clause minimization. *SAT*, 2005:53.

[Turing, 1936] Alan M Turing (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265.

[van der Tak et al., 2012] Peter van der Tak, Marijn JH Heule, and Armin Biere (2012). Concurrent cube-and-conquer. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 475–476. Springer.

[Vander-Swalmen et al., 2011] Pascal Vander-Swalmen, Gilles Dequen, Michaël Krajecki, et al. (2011). Designing a parallel collaborative sat solver. In *17th International Conference on Parallel and Distributed Processing Techniques and Applications, USA, CSREA Press.*

[Wei and Li, 2009] Wanxia Wei and Chu Min Li (2009). Switching between two adaptive noise mechanisms in local search for SAT. *SAT 2009 competitive events booklet: preliminary version*, pages 57–58.

[Wieringa and Heljanko, 2013] Siert Wieringa and Keijo Heljanko (2013). Concurrent clause strengthening. In *Theory and Applications of Satisfiability Testing–SAT 2013*, pages 116–132. Springer.

[Winston, 1992] Patrick Henry Winston (1992). *Artificial intelligence (3rd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Wolper, 2006] Pierre Wolper (2006). *Introduction à la calculabilité: cours et exercices corrigés*. Sciences sup. Dunod.

[Wolpert and Macready, 1997] David H Wolpert and William G Macready (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82.

[Wotzlaw et al., 2012] Andreas Wotzlaw, Alexander Van Der Grinten, Ewald Speckenmeyer, and Stefan Porschen (2012). pfoliouzk: Solver description. *Proc. SAT Challenge*, page 45.

[Zhang et al., 1996] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang (1996). Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560.

[Zhang and Stickel, 2000] Hantao Zhang and Mark Stickel (2000). Implementing the davis–putnam method. *Journal of Automated Reasoning*, 24(1-2):277–296.

[Zhang et al., 2001] Lintao Zhang, Connor Madigan, Matthew W Moskewicz, and Sharad Malik (2001). Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285.

November 26, 2014