Contributions à la résolution parallèle du problème SAT

Vincent Vallade

Thèse présentée pour l'obtention du grade de

Docteur de Sorbonne Université

Spécialité : Ingénierie / Systèmes Informatiques

École doctorale EDITE de Paris (ED130)
Informatique, Télécommunication et Électronique

Laboratoire d'Informatique de Paris 6 (LIP6)

Contributions à la résolution parallèle du problème SAT

Vincent VALLADE

Soutenue le 27 Juin 2023 devant un jury composé de :

Présidente :
Hanna KLAUDEL, Professeure des universités, Université Paris-Saclay, IBISC

Rapporteurs :
Jean-Michel COUVREUR, Professeur des universités, Université d'Orléans, LIFO
Kais KLAI, Maître de conférences, Université Sorbonne Paris Nord, LIPN, CNRS

Examinateur :
Daniel LE BERRE, Professeur des universités, Université d'Artois, CRIL, CNRS

Encadrant :
Julien SOPENA, Maître de conférences, Sorbonne Université, LIP6, CNRS, INRIA

Directeurs :
Souheib BAARIR, Enseignant-Chercheur, EPITA
Fabrice KORDON, Professeur des universités, Sorbonne Université, LIP6, CNRS

Remerciements

Cette partie a pour but d'exprimer ma profonde reconnaissance envers toutes les personnes qui ont joué un rôle essentiel dans mon parcours jusqu'à présent. Il y a cinq ans, obtenir une thèse semblait tout simplement impossible, et cela aurait été le cas sans vous.

Pour commencer, je remercie Kais Klai et Jean-Michel Couvreur d'avoir accepté le rôle de rapporteurs et le temps qu'ils ont consacré à la lecture attentive de mon manuscrit. Merci également à vous Hanna Klaudel et Daniel Le Berre d'avoir accepté d'être membre de mon jury.

Je remercie maintenant l'équipe encadrante de ma thèse pour leur soutien indéfectible tout au long de ces cinq dernières années : Julien Sopena, Souheib Baarir et Fabrice Kordon. J'ai pu admiré vos compétences à la fois techniques et théoriques tout au long de cette thèse. Vous m'avez accompagné dans la conception de nos idées, leur mise en œuvre et leur évaluation, et cette thèse n'aurait pas pû se poursuivre convenablement sans vous.

Je remercie Saeed Nejati et Vijay Ganesh, de l'université de Waterloo, avec qui j'ai pu entretenir une collaboration très fructueuse.

Merci à mes anciens collègues de bureau Ludovic Le Frioux et Hakan Metin, pour m'avoir chaleureusement accueilli dans l'équipe et m'avoir permis de devenir rapidement productive dans mon domaine grâce à vos nombreux conseils et outils. Merci également à mes collègues de bureau : Anissa Kheireddine, Sabrine Saouli et Hao Xu.

Je souhaite également remercier Marc Shapiro, d'avoir été ma porte d'entrée au LIP6 en m'acceptant en tant que stagiaire de master. Je remercie donc particulièrement les encadrants de ce stage Dimitrios Vasilas et Ilyas Toumlilt. Merci au passage aux collègues du master et stagiaires Saalik Hatia, Alexandre Lavigne et Nicolas Guittonneau.

Je souhaite exprimer ma profonde gratitude aux membres du laboratoire, en commençant par les permanents : Luciana Arantes, Swan Dubois, Jonathan Lejeune, Pierre Sens, Béatrice Bérard, Cédric Besse, Claude Dutheillet, Yann Thierry-Mieg et Tewfik Ziadi. Je remercie tous mes collègues, doctorants ou non, pour ces nombreuses discussions et pauses café qui m'ont permis de tenir jusqu'au bout. Merci donc à Aymeric Agon-Rambosson, Maxime Bittan, Antoine Blin, Marjorie Bournat, Damien Carver, Florent Coriat, Cédric Courtaud, Arnaud Favier, Guillaume Fraysse, Yoann Ghigoff, Redha Gouicem, Saalik Hatia, Sara Houhou, Francis Laniel, Gabriel Le Bouder, Mathieu Lehaut, Étienne Le Louët, Célia Mahamdi, Benoît Martin, Darius Mercadier, Sreeja Nair, Ayush Pandey, Baptiste Pires, Laurent Prosperi, Lucas Serrano, Jonathan Sid-Otmane, Ilyas Toumlilt, Dimitrios Vasilas et Daniel Wilhelm.

Et finalement, je remercie ma mère pour son soutien tout au long de mon parcours.

Table des matières

Chapitre

1

Introduction

Cette thèse présente des contributions multiples et orthogonales améliorant la résolution parallèle du problème de satisfaisabilité booléenne ou problème SAT. Une instance d'un tel problème est une formule booléenne de forme particulière (la forme normale conjonctive est la plus courante) représentant, en général, les variables et les contraintes d'un problème du monde réel, tel que la planification multi-contraintes [54], la vérification matérielle et logicielle [17] ou la cryptographie [72]. La résolution du problème SAT consiste à déterminer s'il existe une affectation des variables qui permet de satisfaire la formule. Ce problème a la particularité d'être le premier à être déterminé comme étant NP-complet [30]. Il n'existe donc pas d'algorithme déterministe capable de le résoudre en temps polynomial, mais la validité d'une solution à une instance du problème peut être vérifiée en temps polynomial. Cette définition implique également que tout autre problème NP-complet peut être transformé en un problème SAT en temps polynomial, d'où l'intérêt de résoudre ce problème dans divers domaines. Un algorithme naïf capable de fournir une réponse à ce problème parcourt l'ensemble des combinaisons de valeurs possibles pour chaque variable jusqu'à trouver une combinaison rendant la formule vraie. Si une telle combinaison existe, la formule est SAT, autrement la formule est UNSAT. La complexité exponentielle de cette méthode est évidente et de nombreux algorithmes et heuristiques ont été développés pour accélérer la capacité de résolution de ce problème dans un contexte séquentiel. L'encapsulation de ces algorithmes et heuristiques est appelée un solveur SAT. L'omniprésence de machines multi-coeurs a encouragé des efforts considérables dans la résolution parallèle du problème SAT.

Dans le contexte parallèle, de nombreuses instances de solveurs SAT (potentiellement hétérogènes) sont réparties sur différentes zones de l'espace des combinaisons possibles (appelé sous-espace de recherche) afin d'accélérer la recherche d'une solution. Le partage d'information entre les différents solveurs est un élément clé de l'efficacité. Il est généralement mis en œuvre sous forme d'ensembles de nouvelles contraintes qui sont échangées entre les différents participants à la stratégie de résolution parallèle. Ces contraintes, appelées clauses, prennent la forme d'une disjonction de littéraux (une variable ou sa négation), et sont utilisées pour lier logiquement plusieurs variables. Lorsqu'un solveur détermine qu'un sous-espace de recherche

ne contient pas de solution, cette information est enregistrée sous forme de clause. De manière abstraite, cette clause encode un chemin qui mène à un espace sans solution. Une affectation de variables qui rend cette clause fausse permettra au solveur de se rendre compte plus rapidement que le chemin actuel est une impasse. Les algorithmes capables d'apprendre ces clauses dynamiquement sont appelés algorithme (ou solveur) Conflict-Driven Clause Learning (CDCL) [69, 87]. Bien que la stratégie de résolution parallèle tente au mieux de répartir les différents solveurs sur des sous-espaces de recherche différents, une clause apprise pour un sous-espace S_1 peut permettre d'éliminer des mauvais chemins dans un sous-espace S_2. Le partage de ces clauses augmente donc considérablement la capacité de résolution, que ce soit pour déterminer la satisfaisabilité ou l'insatisfaisabilité d'une formule. Dans un premier temps, nos travaux se sont appliqués à penser de nouvelles stratégies pour améliorer le partage dans un solveur SAT parallèle. Nous avons dérivé deux contributions sur le sujet : une méthode permettant de sélectionner l'information à partager et une méthode permettant de minimiser l'information partagée.

Un deuxième point important dans l'efficacité de la résolution parallèle est une division correcte du travail. Si la division de l'espace de recherche fournie aux différents solveurs n'est pas faite de manière intelligente, nous pouvons avoir plusieurs solveurs effectuant le même travail où avoir tous les solveurs coincés dans des sous-espaces difficiles limitant le bénéfice de la parallélisation. Une dernière contribution consiste à étudier les performances d'une méthode reposant sur un algorithme probabiliste pour initialiser et guider les différents fils d'exécution dans l'espace de recherche.

CONTRIBUTIONS

Une politique de partage basée sur la structure des communautés (Chapitre 3).

Cette contribution vise à améliorer le partage d'informations entre les solveurs. Le développement d'une politique de partage peut être résumé par "le problème de l'identification de clauses apprises de haute qualité". Ces clauses, lorsqu'elles sont partagées entre les différents solveurs d'une stratégie de résolution parallèle, devraient conduire à une amélioration des performances. Le terme "clause de haute qualité" est souvent défini en termes de métriques que les concepteurs de solveurs ont identifiées au cours d'années d'études empiriques. Certaines métriques ont émergé en termes de performance, mais dépendent fortement de l'état local de leur solveur d'origine. La valeur de cette métrique une fois que la clause est importée par un autre solveur séquentiel peut être complètement incohérente par rapport à sa progression.

Dans ce chapitre, nous proposons et évaluons une nouvelle métrique visant à identifier les clauses apprises de haute qualité et une politique concomitante de partage des clauses basée sur la combinaison d'une métrique locale et la structure des communautés exposée par les formules SAT. Nous appelons un groupe de variables avec un lien fort entre elles et un lien faible avec le reste des variables une communauté. Une instance du problème SAT peut contenir des

dizaines ou des milliers de communautés. Cette dernière est une propriété fréquente des instances encodant des problèmes du monde réel. Il est bien connu que les instances industrielles (en opposition à celles générées aléatoirement) ont des variables plus contraintes ensemble (liées par plus de clauses). Le concept de structure des communautés a été proposé comme une explication possible pour l'extraordinaire performance des solveurs SAT sur des instances industrielles. De ce fait, cette structure est une candidate naturelle pour servir de base à une métrique permettant d'identifier les clauses de haute qualité.

Cette contribution a été publiée dans la 23$^{\text{rd}}$ *International Conference on Theory and Applications of Satisfiability Testing – SAT2020* [95].

Intégration et évaluation d'une stratégie de minimisation de clauses (Chapitre 4).

Si la contribution précédente tente d'améliorer le partage d'informations avec une meilleure sélection des clauses, cette contribution va augmenter l'efficacité du partage en réduisant la taille de l'information partagée. En effet, il est bien connu que plus les clauses apprises sont courtes, plus leur capacité à réduire l'espace de recherche est élevée. En effet, une petite clause nécessite moins de propagations pour être satisfaite ou mise en conflit. Ceci explique la proposition de différentes techniques basées sur la résolution pour les raccourcir, un processus connu sous le nom de minimisation.

Cette contribution se décompose en deux parties :

— Présenter et évaluer une stratégie de minimisation pour solveurs SAT parallèles, inspirée d'une version séquentielle d'un algorithme existant.

— Combiner cette stratégie avec différentes politiques de partage existante pour améliorer la qualité du partage.

La première partie de cette contribution a été publiée sous la forme d'un papier dans le 12$^{\text{th}}$ *NASA Formal Methods Symposium – NFM2020* [97]. Cette même contribution implémentée dans un solveur appelé `P-mcomsps` a permis de remporter la médaille d'or des compétitions SAT parallèles 2020 [94] et 2021 [98].

La deuxième partie de cette contribution a été publiée dans la 14$^{\text{th}}$ *Pragmatics of SAT international workshop – POS2023*

Diversifier un solveur parallèle grâce au Bayesian Moment Matching (Chapitre 5).

Ce chapitre présente notre dernière contribution, qui va cette fois améliorer la division du travail. Nous basant sur des travaux permettant d'améliorer le démarrage de la résolution, nous utilisons le même mécanisme pour guider le solveur en cours de résolution.

Dans leur article, Duan et al.[35] présentent un algorithme d'apprentissage probabiliste basé sur le Bayesian Moment Matching (BMM) et utilisé comme préprocesseur pour un solveur SAT. Le but de cet algorithme est de fournir une affectation initiale à partir de laquelle la recherche du solveur peut commencer. Dans ce chapitre, nous présentons et évaluons une technique basée sur BMM appliquée au cours de la résolution pour paramétrer la routine de décision (routine qui détermine quelle sera la prochaine variable à affecter et à quelle valeur) d'un solveur. En particulier, le comportement initialement aléatoire de ce mécanisme est exploité dans le développement d'un solveur parallèle (en utilisant une graine différente pour initialiser la distribution de probabilité pour chaque solveur). Ce nouveau solveur parallèle s'est montré performant sur des instances de problèmes cryptographiques.

Cette contribution a été publiée sous la forme d'un short papier en 2022 dans le cadre du *Symposium on Dependable Software Engineering – SETTA2022* [96].

Chapitre

2

Le problème de satisfiabilité booléenne

Contents

Ce chapitre introduit les notions nécessaires pour appréhender les problématiques abordées dans ce manuscrit, qui s'appuie sur les nombreux travaux réalisés sur la résolution du problème de satisfaisabilité booléenne ou problème SAT. Les instances de ce problème représentent des systèmes de contraintes sous la forme de formules propositionnelles. Résoudre ce problème consiste à déterminer s'il est possible de satisfaire toutes les contraintes imposées par un système de contrainte donné et, le cas échéant, exhiber une solution . Un algorithme capable de fournir une réponse à ce problème est appelé un solveur SAT. Ce type de solveur a

été utilisé pour déterminer la validité de systèmes dans de nombreux domaines tels que la planification multi-contraintes [54], la vérification matérielle et logicielle [17], la biologie [67] ou encore la cryptographie [72]. De nombreux algorithmes et heuristiques ont été développés pour accélérer la capacité de résolution de ce problème, principalement dans un contexte séquentiel. L'omniprésence de machines multi-coeurs a encouragé des efforts considérables dans la résolution parallèle du problème SAT [12]. Cette thèse s'inscrit dans le prolongement de ces efforts et va présenter multiples contributions dans ce domaine.

Structure du chapitre. Dans ce chapitre, nous commençons par introduire les bases de la logique propositionnelle (Section 2.1). Puis, nous définissons plus en détail le problème SAT et les principaux algorithmes et heuristiques conçus pour le résoudre en séquentiel (Section 2.2). Finalement, nous présenterons les problématiques principales de cette thèse, la résolution parallèle du problème SAT (Section 2.3).

2.1 REPRÉSENTATION BOOLÉENNE DU PROBLÈME

La logique propositionnelle permet la construction de raisonnement en reliant des variables propositionnelles à l'aide d'opérations. Nous allons détailler ici le langage utilisé tout au long de ce manuscrit.

2.1.1 *Syntaxe et sémantique de la logique propositionnelle*

— constantes : $\top$ et $\bot$ représentant vrai et faux respectivement.

— variables propositionnelles : une variable est un identifiant associé à une valeur, $\top$ ou $\bot$. Exemples de variables valides : var, a, ou x_3 ;

— ponctuation : parenthèse ouvrante et fermante '(' et ')' ;

— opérations : $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, et $\oplus$ respectivement appelée négation, disjonction, conjonction, implication, équivalence et disjonction exclusive.

En utilisant ce vocabulaire, nous pouvons décrire les règles pour construire une formule propositionnelle. Une formule peut être vue comme une liste de symboles du vocabulaire.

La signification de ces opérateurs est donnée dans la Table 2.1.

x	y	$\neg x$	$x \vee y$	$x \wedge y$	$x \Rightarrow y$	$x \Leftrightarrow y$	$x \oplus y$
0	0	1	0	0	1	1	0
0	1	1	1	0	1	0	1
1	0	0	1	0	0	0	1
1	1	0	1	1	1	1	0

TABLE 2.1 – Table de vérité des opérateurs

Pour évaluer une formule, nous donnons pour chaque variable une valeur dans un ensemble $\mathbb{B} = \{\text{true}, \text{false}\} = \{\top, \bot\} = \{1, 0\}$.

La valeur de la formule est alors obtenue en remplaçant les variables par leurs valeurs en appliquant les règles définies dans la Table 2.1. Afin de raisonner sur des formules, nous définissons formellement ici les différents concepts dont nous avons besoin.

Définition 1: Littéral

Un *literal* l est une variable ou sa négation. Pour une variable donnée x, le littéral positif est représenté par x et celui négatif par $\neg x$.

Nous désignons $\mathcal{V}_\varphi$ ($\mathcal{L}_\varphi$) comme étant l'ensemble de variables (littéraux) utilisées dans φ (l'index dans $\mathcal{V}_\varphi$ et $\mathcal{L}_\varphi$ est généralement omis lorsqu'il est clair dans le contexte).

La valeur donnée aux différentes variables d'une formule est appelée *affectation*. Une affectation est une fonction telle que définie par Définition 2.

Définition 2: Affectation

Pour une formule φ donnée, nous définissons une *affectation* des variables de φ, noté α, comme la fonction $\alpha : \mathcal{V}_\varphi \to \mathbb{B}$, qui associe une valeur booléenne à certaines ou à toutes les variables de φ.

Pour une formule φ donnée, une affectation α_p est *partielle* si une ou plusieurs variables de φ n'ont pas d'image par α_p. Au contraire, α_c est *complète* si toutes les variables de φ ont une image par α_c.

Pour illustrer ces définitions, nous présentons différents exemples d'affectations et précisons si elles sont partielles ou complètes.

Exemple: Soit $\varphi = (x \wedge y) \vee \neg z$ une formule propositionnelle et $\alpha_1 = \{x \mapsto 0, y \mapsto 1, z \mapsto 0\}$ une affectation des variables de φ. Cette affectation peut aussi être notée $\alpha_1 = \{\neg x, y, \neg z\}$. De plus, α_1 est complète. L'affectation $\alpha_2 = \{x, \neg z\}$ est partielle, car α_2 n'est pas définie pour la variable z.

Nous avons vu comment les variables d'une formule propositionnelle donnée peuvent avoir une valeur d'après une affectation. Nous voulons maintenant extraire une valeur pour une formule selon la valeur de ses variables. Définition 3 donne les règles pour calculer inductivement la valeur d'une formule, selon une affectation de ses variables.

Définition 3: Valeur d'une formule propositionnelle

Soit φ une formule propositionnelle et α une affectation des variables de φ. $[\varphi]_\alpha$ désigne la valeur de la formule φ pour l'affectation α. La valeur de $[\varphi]_\alpha$ est définie de manière récursive. Soit les formules φ et ψ, x une variable et α une affectation, nous avons :

- $[\top]_\alpha = 1$, $[\bot]_\alpha = 0$;
- $[x]_\alpha = \alpha(x)$;
- $[\neg\varphi]_\alpha = 1 - [\varphi]_\alpha$;
- $[(\varphi \vee \psi)]_\alpha = max\{[\varphi]_\alpha, [\psi]_\alpha\}$;
- $[(\varphi \wedge \psi)]_\alpha = min\{[\varphi]_\alpha, [\psi]_\alpha\}$;
- $[(\varphi \Rightarrow \psi)]_\alpha =$ si $[\varphi]_\alpha$ alors 1 sinon $[\psi]_\alpha$;
- $[(\varphi \Leftrightarrow \psi)]_\alpha =$ si $[\varphi]_\alpha$ est égale à $[\psi]_\alpha$ alors 1 sinon 0 ;
- $[(\varphi \oplus \psi)]_\alpha =$ si $[\varphi]_\alpha$ est égale à $[\psi]_\alpha$ alors 0 sinon 1.

La valeur d'une formule dépend de la valeur de ses variables et des relations entre celles-ci établies à travers les différents opérateurs logiques. Ainsi, l'évaluation d'une formule peut aussi être présentée sous la forme d'une *table de vérité*. La table de vérité d'une formule φ est la table représentant les différentes valeurs possibles de φ selon les valeurs affectées à ses variables. Chaque colonne correspond à une valeur possible pour une variable ou pour la valeur de φ correspondante. Une formule φ composée de n variables (*i.e.*, $|\mathcal{V}| = n$) a 2^n affectations possibles, chacune représentée par une ligne dans la table de vérité.

La table de vérité des différents opérateurs logiques du vocabulaire est présentée dans la Table 2.1. De plus, la Table 2.2 est un exemple de table de vérité pour les formules : $x \wedge \neg y$, $\neg(x \Rightarrow y)$, $x \oplus x$ et $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$.

x	y	$x \wedge \neg y$	$\neg(x \Rightarrow y)$	$x \oplus x$	$(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$
0	0	0	0	0	1
0	1	0	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1

TABLE 2.2 – Exemple de table de vérité

Nous sommes maintenant en mesure d'évaluer la valeur d'une formule pour toutes les affectations de ses variables. Selon la valeur prise par une formule pour une affectation particulière, nous appelons cette affectation *modèle* ou *contre-modèle* de la formule.

Définition 4: Modèle/Contre-modèle de la formule

Pour une formule φ donnée, si une affectation complète α donne la valeur 1 (respectivement 0) à φ, *i.e.,* $[\varphi]_\alpha = 1$ (respectivement $[\varphi]_\alpha = 0$), nous disons que cette affectation α est un *modèle* (respectivement *contre-modèle*) de la formule φ.

L'exemple suivant présente un modèle et un contre-modèle de quelques formules de la Table 2.2.

Exemple: L'affectation $\{x, \neg y\}$ est un modèle de la formule $x \wedge \neg y$, tandis que l'affectation $\{\neg x, y\}$ est un contre-modèle de la formule $\neg(x \Rightarrow y)$.

Selon les valeurs possibles pour la formule, une formule peut être *satisfaisable* ou *insatisfaisable*. Définition 5 présente ces deux notions.

Définition 5: Formule satisfaisable/insatisfaisable

Une formule est *satisfaisable* (respectivement *insatisfaisable*) si il existe au moins une (respectivement aucune) affectation de ses variables formant un modèle.

Cas spécial : Une formule pour laquelle il n'existe pas d'affectation de ses variables formant un contre-modèle est appelée une *tautologie*.

L'exemple suivant est issu de la table de vérité donnée par la Table 2.2, il présente différentes formules et identifie celles qui sont satisfaisable, insatisfaisable ou une tautologie. Dans le reste de ce document, nous noterons ces deux caractéristiques SAT et UNSAT respectivement.

Exemple: Les formules $x \wedge \neg y$, $\neg(x \Rightarrow y)$ et $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$ sont SAT. Comme montré dans la Table 2.2, il existe au moins une affectation (une ligne dans la table) pour laquelle une valeur de la formule est 1. Par exemple, $\{x, \neg y\}$ est une solution possible pour ces trois formules. La formule $x \oplus x$ est UNSAT, pour chaque affectation la valeur de la formule est 0. Finalement, $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$ est une tautologie : pour toute affectation, la valeur de la formule est 1.

Il est nécessaire de pouvoir confronter deux différentes formules dans le but de déterminer si il existe un lien (logique) entre elles. Dans ce but, nous définissons ici les concepts de *conséquence logique* et d'*équivalence logique* définie respectivement par Définition 6 et Définition 7.

Définition 6: Conséquence logique

Soit φ et ψ deux formules propositionnelles, ψ est une conséquence logique de φ si tout modèle de φ est un modèle pour ψ. Cette conséquence logique est notée : $\varphi \vDash \psi$.

Définition 7: Equivalence logique

Deux formules φ et ψ sont équivalentes si elles ont la même valeur pour chaque affectation. Cette équivalence logique est notée : $\varphi \equiv \psi$.

L'exemple suivant compare les formules présentées dans la Table 2.2.

Exemple: Soit les deux formules $\varphi_1 = x \wedge \neg y$ et $\psi_1 = (x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$. φ_1 est une conséquence logique de la formule ψ_1. En effet, nous pouvons observer dans la Table 2.2 que chaque modèle de φ_1 est un modèle de ψ_1. Nous pouvons donc écrire $\varphi_1 \vDash \psi_1$. Les formules $\varphi_2 = x \wedge \neg y$ et $\psi_2 = \neg(x \Rightarrow y)$ sont logiquement équivalentes ($\varphi_2 \equiv \psi_2$), elles ont la même valeur pour chaque affectation.

2.1.2 *Forme normale*

Nous avons défini les bases de la logique propositionnelle nécessaire pour comprendre la représentation utilisée pour le problème SAT. Les formules propositionnelles représentant ce problème (appelée *instance* du problème), ont une structure particulière, appelée *forme normale*. Afin de définir correctement la forme normale qui nous intéresse, nous devons d'abord introduire les concepts de *cube* et de *clause*.

Définition 8: Cube

Un cube γ est une conjonction finie de k littéraux représentée par la formule $\gamma = \bigwedge_{i=1}^{k} \ell_i$, ou par l'ensemble de ses littéraux $\gamma = \{\ell_i\}_{i \in [\![1,k]\!]}$.

Définition 9: Clause

Une clause ω est une disjonction finie de k littéraux représentée par la formule $\omega = \bigvee_{i=1}^{k} \ell_i$, ou par l'ensemble de ses littéraux $\omega = \{\ell_i\}_{i \in [\![1,k]\!]}$.

On dit qu'une clause est *unitaire*, *binaire*, *ternaire* ou n-*aire* si elle contient respectivement une, deux, trois ou n ($\in \mathbb{N}^*$) littéraux.

Définition 10: Clause complémentaire

La clause complémentaire d'une clause de taille k, $\omega = \{\ell_i\}_{i \in [\![1,k]\!]}$ est la clause contenant la négation de tous les littéraux de ω notée $\overline{\omega} = \{\neg \ell_i\}_{i \in [\![1,k]\!]}$

Définition 11: Clause Subsumée

Soit deux clauses ω_1 et ω_2, on dit que ω_1 subsume par ω_2 si tous les littéraux de ω_1 sont contenus dans ω_2. En d'autres termes, nous avons $\omega_1 \subseteq \omega_2$.

L'exemple suivant illustre les différents types de clauses présentés et analyse leurs particularités.

Exemple: La clause $\omega_1 = \{\neg a\}$ est unitaire. La clause $\omega_2 = \{a, \neg b\}$ est binaire. La clause $\omega_4 = \{\neg a, \neg b, \neg c, \neg e, \neg f\}$ est une clause 5-aire. De plus, nous pouvons observer que ω_4 est subsumée par ω_1.

Avec ces définitions, nous pouvons finalement présenter une forme normale particulière qui est la *forme normale conjonctive*.

Définition 12: Forme Normale Conjonctive

Une formule propositionnelle en forme normale conjonctive (ou *CNF* pour *Conjunctive Normal Form*) φ est une conjonction de clauses. Une formule CNF peut être représentée par la formule $\varphi = \bigwedge_{i=1}^{k} \omega_i$ ou par l'ensemble de ses clauses $\varphi = \{\omega_i\}_{i \in [\![1,k]\!]}$.

Par exemple la formule propositionnelle suivante est en forme normale conjonctive : $\varphi = (a \vee b \vee \neg c) \wedge (\neg a) \wedge (\neg d \vee \neg b)$

Ces définitions nous permettent d'expliquer le problème de satisfaisabilité booléenne (ou problème SAT), comme étant le problème consistant à **déterminer si une formule propositionnelle (traditionnellement en forme normale conjonctive) est satisfaisable ou non.** Ce problème a une grande importance dans le domaine de la théorie de la complexité, car il est le premier à être prouvé NP-complet [30]. Nous définissons ici les termes nécessaires pour comprendre cette caractéristique. La théorie de la complexité classe des problèmes algorithmiques selon des contraintes comme le temps de calcul ou l'espace mémoire. Nous présentons ici quelques-unes de ces classes pour comprendre où se situe le problème SAT selon la théorie de la complexité :

— Un problème qui peut être résolu par une machine de Turing non déterministe en temps polynomial fait partie de la classe "Non déterministe polynomial" ou **NP** ;

— Un problème p est **NP-difficile** si tout problème p' dans NP peut être réduit en temps polynomial à p ;

— Un problème est **NP-complet** si il appartient aux classes NP et NP-difficile.

Il n'existe pas d'algorithmes déterministes réellement efficaces pour résoudre les problèmes NP-complets, car leur complexité (en temps de calcul) est exponentielle par rapport aux paramètres d'entrée. Cependant, la validité d'une solution donnée peut-être vérifiée en temps polynomial. En raison des propriétés susdécrites, tous les problèmes NP-complets peuvent

être ramenés au problème SAT. C'est une des raisons pour laquelle le développement de programme (appelé *solveur SAT*) pouvant résoudre une instance de ce problème est devenu un enjeu industriel et académique majeur.

Un algorithme trivial pour un solveur SAT consiste à énumérer toutes les affectations possibles des variables de la formule, jusqu'à trouver un modèle pour celle-ci ou déterminer son insatisfaisabilité après avoir mis en échec toutes les affectations. Évidemment, le nombre d'affectations complètes générées est exponentiel en fonction du nombre de variables de la formule. Des décennies de recherche ont permis de développer de nombreux paradigmes et heuristiques pour accélérer la vitesse de résolution des solveurs SAT. La section suivante présente la forme d'un algorithme de résolution standard dans les solveurs SAT modernes.

2.2 RÉSOLUTION SÉQUENTIELLE DU PROBLÈME SAT

Les algorithmes déterministes modernes de résolution du problème SAT n'itèrent pas sur des affectations complètes, mais plutôt sur les littéraux de la formule. À chaque itération, ils vont procéder à un mécanisme de simplification sur la formule. Soit la formule φ et le littéral $\ell \in \mathcal{L}$. La simplification de φ par ℓ est définie par : $\varphi|_\ell = \{\mathcal{C}\backslash\{\neg\ell\} \mid \mathcal{C} \in \varphi \wedge \neg\ell \in \mathcal{C}\} \cup \{\mathcal{C} \mid \mathcal{C} \in \varphi \wedge \ell \notin \mathcal{C} \wedge \neg\ell \notin \mathcal{C}\}$. Le premier terme de l'union correspond aux clauses ne contenant $\neg\ell$ (dans lesquelles $\neg\ell$ a été supprimé), le second terme correspond aux clauses contenant ni ℓ ni $\neg\ell$. Plus généralement, la simplification de la formule φ par un ensemble de littéraux est définie par : $\varphi|_{\{\ell_1,\ell_2,\ldots,\ell_n\}} = (\ldots((\varphi|_{\ell_1})|_{\ell_2})\ldots|_{\ell_n})$. L'ordre des simplifications n'est pas important et mène à la même formule.

Si la simplification d'une formule mène à l'apparition d'une clause vide, on dit que l'algorithme est arrivé à un *conflit*, l'affectation courante est considérée comme invalide et l'algorithme va revenir sur une ou plusieurs itérations afin de revenir à une affectation partielle ne rendant pas la formule UNSAT. Au contraire, si la clause vide n'est pas dérivée par la simplification, l'algorithme procède à l'itération suivante. L'un des premiers algorithmes non intensifs en mémoire pour résoudre le problème SAT est l'algorithme Davis Putnam Logemann Loveland (DPLL) [32], décrit dans la sous-section suivante.

2.2.1 *L'algorithme Davis Putnam Logemann Loveland (DPLL)*

Cet algorithme, en itérant sur les littéraux de la formule, effectue un parcours en profondeur sur un arbre binaire dont les nœuds sont des formules simplifiées. Une feuille de l'arbre contenant une clause vide indique que la branche correspondante est un contre-modèle de la formule. Au contraire, une feuille correspondant à la formule vide indique que toutes les clauses ont été satisfaites, la branche courante est donc un modèle de la formule. On appelle communément l'ensemble des sous-arbres en cours d'exploration ou pas encore exploré *l'espace de recherche* du solveur. Si toutes les feuilles de l'arbre correspondent à des contre-modèles, la formule est UNSAT. Si au moins une feuille est un modèle pour la formule, celle-ci est SAT. Algorithme 1 présente cet algorithme, ici sous sa forme récursive.

Chaque appel commence par l'exécution d'une routine appelée `propagation unitaire` (Ligne 4) dont l'implémentation est détaillée dans l'Algorithme 2. Cette routine cherche toutes les clauses unitaires dans la formule φ passée en paramètre. Puis, pour assurer la satisfaisabilité, ajoute ses littéraux à l'affectation courante. φ est ensuite simplifiée par l'ensemble de ces littéraux comme expliqué précédemment. La procédure se termine lorsque toutes les clauses unitaires ont été traitées ou qu'un conflit a été constaté (la simplification dérive la clause vide). Elle retourne la formule simplifiée et la nouvelle affectation.

DPLL retourne au niveau récursif précédent si la `propagation unitaire` dérive la formule vide ou une formule contenant une clause vide (Lignes 6 et 8). En effet, l'algorithme DPLL ne revient que d'un seul niveau de récursion après avoir atteint un conflit.

La descente de l'algorithme dans l'espace de recherche se fait à travers un mécanisme appelé *décision* ou *branchement*. Ces termes désignent l'action de déterminer quelle sera la prochaine variable à affecter et avec quelle valeur (Ligne 9). Dans le pseudo-code présenté, cette décision est rajoutée au problème au sein des appels récursifs sous forme d'une clause unitaire (Ligne 10).

```
1  fonction DPLL(φ : formule CNF, α : affectation)
2      retourne ⊤ si φ est SAT et ⊥ autrement
3      (φ', α') ← propagationUnitaire(φ|_α)
4      α ← α ∪ α'                          // Ajouter les littéraux propagés dans α
5      si φ' = ∅ alors
6          retourner ⊤                                  // φ est SAT
7      si ∅ ∈ φ' alors                                  // Il y a un conflit
8          retourner ⊥                                  // La branche est UNSAT
9      x ← choixLitteralBranchement()
10     retourner DPLL(φ ∪ {x}, α) ou DPLL(φ ∪ {¬x}, α)
```

Algorithme 1 : L'algorithme DPLL.

```
1  fonction propagationUnitaire(φ : formule CNF, α : affectation)
2      retourne formule CNF et affectation α
3      tant que {l} ∈ φ et {} ∉ φ faire
           // Supprime toutes les clauses contenant l et tous les littéraux ¬l
4          φ ← φ |_l
5          α ← α ∪ {l}
6      retourner φ, α
```

Algorithme 2 : La propagation unitaire.

On note comme différence avec l'algorithme naïf le fait qu'un conflit permet de couper court à la création de l'affectation. En pratique, un solveur n'a pas besoin de générer toutes les affectations possibles pour prendre la décision SAT ou UNSAT. L'efficacité de DPLL est fortement

dépendante de l'implémentation de la procédure `decisionLitteral`. L'objectif de cette fonction est de trouver un littéral qui va générer le maximum de propagations unitaires. S'il n'est virtuellement pas possible de trouver la variable optimale parce qu'il s'agit d'un problème NP-difficile, il existe différentes heuristiques pour effectuer ce choix, certaines seront présentées dans Section 2.2.5. Parmi les défauts de cette approche figure le fait qu'il n'existe aucun mécanisme pour empêcher le solveur de reproduire la même séquence de décisions et de propagations conflictuelles à différents endroits de l'espace de recherche. En effet, certaines affectations partielles conduiront toujours à un conflit, quelles que soient les affectations précédentes. La propagation unitaire est une procédure très coûteuse et donc reproduire un travail inutile est un gâchis de ressources CPU. Par extension, un mécanisme permettant de décrire finement la raison d'un conflit permet également de revenir sur plus d'un niveau de décision, accélérant encore la résolution. Suivant ces principes, la communauté de recherche s'est attaquée au développement de mécanismes d'*apprentissage*, un mécanisme permettant au solveur de se souvenir de ses précédentes erreurs. L'algorithme d'*apprentissage de clauses par conflits* ou *conflict-driven clause learning (CDCL)* implémente ce mécanisme qui permettra à la résolution du problème SAT d'être applicable à de larges problèmes définissant des contraintes du monde réel.

2.2.2 *Conflict-Driven Clause Learning*

L'Algorithme 3 présente une vue de haut niveau de l'algorithme CDCL (ici, l'implémentation de l'algorithme est sous une forme itérative). Comme indiqué précédemment, l'algorithme peut revenir sur plus d'une décision, ainsi on introduit une nouvelle variable lvl qui indique le niveau de décision courant (Ligne 3).

Cet algorithme est basé sur une boucle principale qui a une structure similaire à l'algorithme DPLL. En effet, on commence par une `propagationUnitaire` (Ligne 5), dont le résultat peut déterminer la satisfaisabilité de la formule (la formule simplifiée est vide Ligne 8). La différence se trouve dans le cas où la `propagationUnitaire` dérive la clause vide (Ligne 12). Dans ce cas, deux scénarios sont possibles : (i) l'algorithme n'a pris aucune décision (ou est revenu sur toutes ses décisions), il retourne donc $\bot$ (Ligne 11) ; (ii) autrement, l'algorithme déduit la suite de décisions et propagations à l'origine de ce conflit dans une procédure appelée `analyseDeConflit` que nous décrirons dans la prochaine sous-section. Pour le moment nous indiquons que cette procédure produit une clause ω résumant le conflit. Cette clause est ensuite *apprise* (ajoutée aux autres clauses de la formule) par le solveur. Par la suite, nous appellerons *clauses apprises* ces clauses déduites par l'analyse de conflit. Si cette clause apprise va permettre au solveur d'éviter de prendre ultérieurement la même mauvaise séquence de décision, elle ne modifie pas le problème original pour autant. Autrement dit $\varphi \equiv \varphi \cup \{\omega\}$.

Un autre apport de ces clauses apprises est qu'elles peuvent être utilisées pour calculer un point de retour plus intéressant que le simple renversement de la décision précédente. C'est le rôle de la procédure `calculNiveauDeDecision` (Ligne 14). Nous appelons l'action de revenir à un niveau de décision particulier, annulant au passage les affectations et simplifications dues

```
1  fonction CDCL(φ : formule CNF)
       /* retourne ⊤ si φ est SAT, ⊥ sinon (UNSAT)                          */
2      α ← ∅                                              // affectation courante
3      lvl ← 0                                            // niveau de décision courant
4      Indéfiniment
5          (φ′, α′) ← propagationUnitaire(φ|α)
6          α ← α ∪ α′                                     // Ajouter les littéraux propagés dans α
7          si φ′ = ∅ alors
8              retourner ⊤                                // φ est SAT
9          si ∅ ∈ φ′ alors                                // Il ya un conflit à analyser
10             si lvl = 0 alors
11                 retourner false                        // φ est UNSAT
12             ω ← analyseDeConflit(φ, α)
13             φ ← φ ∪ {ω}
14             lvl ← calculNiveauDeDecision(lvl, ω, ...)
15             α ← {ℓ ∈ α | δ(ℓ) ≤ lvl}
16         sinon
17             α ← α ∪ {choixLitteralBranchement()}       // Choix d'un nouveau
                littéral de branchement
18             lvl ← lvl + 1
```

Algorithme 3 : Conflict-driven clause learning algorithm (CDCL)

aux propagations, *backjump* et cette action est effectuée Ligne 15. Dans cette ligne, chaque littéral de l'affectation actuelle dont le niveau de décision, extrait avec $\delta(\ell)$, est supérieur au nouveau niveau de décision est supprimé de l'affectation.

Finalement, s'il n'y a pas de conflit, un nouveau littéral est sélectionné pour progresser dans la résolution de φ. Le niveau de décision est donc incrémenté en conséquence (Lignes 17-18).

2.2.3 *Analyse de conflit*

Il y a un conflit quand l'algorithme rencontre une situation qui implique qu'une variable soit simultanément $\top$ et $\bot$ (*i.e.,* l'affectation courante est inconsistante et une clause est devenue vide). En faisant la même décision/propagation, un conflit peut apparaître plusieurs fois durant la résolution. Pour éviter cette répétition, les raisons d'un conflit doivent être identifiées et exploitées pour guider les futures décisions du solveur. Cela se fait en construisant et en analysant ce que nous appelons un *graphe d'implication*. Le schéma suivant est utilisé pour produire une *clause apprise* : si $\bigwedge_{i=1}^{k} \ell_i$ est l'origine d'un conflit, alors $\bigvee_{i=1}^{k} \neg\ell_i$ est une condition nécessaire pour éviter ce conflit. Puisque cette information est déjà contenue dans la formule originale, cette clause apprise peut être ajoutée sans risque à la formule, et supprimée par la suite. Soit la formule (extraite de [19]) :

$$\begin{aligned}\varphi &= \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\} \\ &= \{\{x_1, x_2, \neg x_3\}, \{x_1, \neg x_4\}, \{x_3, x_4, x_5\}, \{\neg x_5, \neg x_6\}, \{x_7, \neg x_5, \neg x_8\}, \{x_6, x_8\}\}\end{aligned}$$

La Figure 2.3 présente la construction d'un graphe d'implication au fur et à mesure des décisions prises par un solveur CDCL. Les nœuds colorés représentent des décisions, les autres nœuds sont des propagations. Le label d'un nœud $x_i@k$ représente une décision/propagation de la variable x_i au niveau de décision k. Les arcs entre les nœuds sont marqués de la clause qui est à l'origine de la propagation selon l'affectation courante. Le solveur prend donc dans l'ordre les décisions suivantes : $\neg x_7$, $\neg x_2$, $\neg x_1$. Les deux premières décisions n'entraînent pas de propagation immédiate, mais la dernière décision provoque une séquence de propagations qui aboutit à un conflit : selon l'affectation courante, la variable x_8 doit être simultanément $\top$ et $\bot$. La raison derrière ce conflit peut-être représentée par le cube $\{\neg x_7, \neg x_2, \neg x_1\}$. Selon la formule précédente, le solveur peut apprendre la clause $\{x_7, x_2, x_1\}$ afin d'éviter ce conflit dans la suite de la résolution.

En pratique, le graphe d'implication est davantage exploité par le concept d'*Unit Implication Point (UIP)* [87]. Un UIP, pour le niveau de décision où le conflit apparaît, est un nœud qui se trouve sur chaque chemin allant de la décision au conflit. Il peut exister plusieurs IUP, mais il y en a nécessairement au moins un, car le nœud de décision est un UIP. Dans le cas où il y en a plusieurs, les UIP sont classés en fonction de leur distance par rapport au conflit, le 1^{er}-UIP est donc le plus proche du conflit. Un UIP divise le graphe d'implication en deux parties le *côté raison* contient les variables de décision responsables de la contradiction et le *côté conflit* contient le conflit. Pour construire la clause de conflit, il faut extraire le cube de littéraux ayant un arc qui traverse les deux sections du graphe. S'il existe des techniques pour calculer tous les IUP en temps linéaire, et donc sans trop de surcoûts à la procédure d'analyse de conflit, il est connu que le 1^{er}-UIP permet d'extraire la plus petite clause de conflit [102]. Dans la Figure 2.3, ce cube pour le 1^{er}-UIP est $\{\neg x_7, x_5\}$. La clause de conflit à apprendre est donc $\{x_7, \neg x_5\}$ et elle représente deux niveaux de décision, 1 et 3. Une fois cette clause ajoutée à la base de clauses apprises, le solveur va revenir au niveau de décision précédent le conflit, ici 1 (le niveau de décision 2 n'est pas présent dans la clause). On dit que la clause apprise est *assertive* car l'IUP est la seule variable sans valeur après le backjump. En conservant la décision $\neg x_7$, la clause apprise va forcer la propagation de la valeur $\bot$ pour la variable x_5, évitant ainsi ce conflit.

2.2.4 *Back-jumping / Redémarrage*

Dans l'algorithme DPLL, lorsqu'un conflit est détecté, un retour en arrière (au niveau de décision précédent) est effectué. Or, l'analyse des conflits permet de calculer des sauts arrière, qui peuvent être plus longs que de simples retours au niveau de décision précédent, et qui représentent potentiellement de meilleurs points de départ pour poursuivre la recherche d'une solution. Plus précisément, nous devons faire un saut arrière jusqu'au niveau le plus élevé représenté dans la clause nouvellement apprise [69].

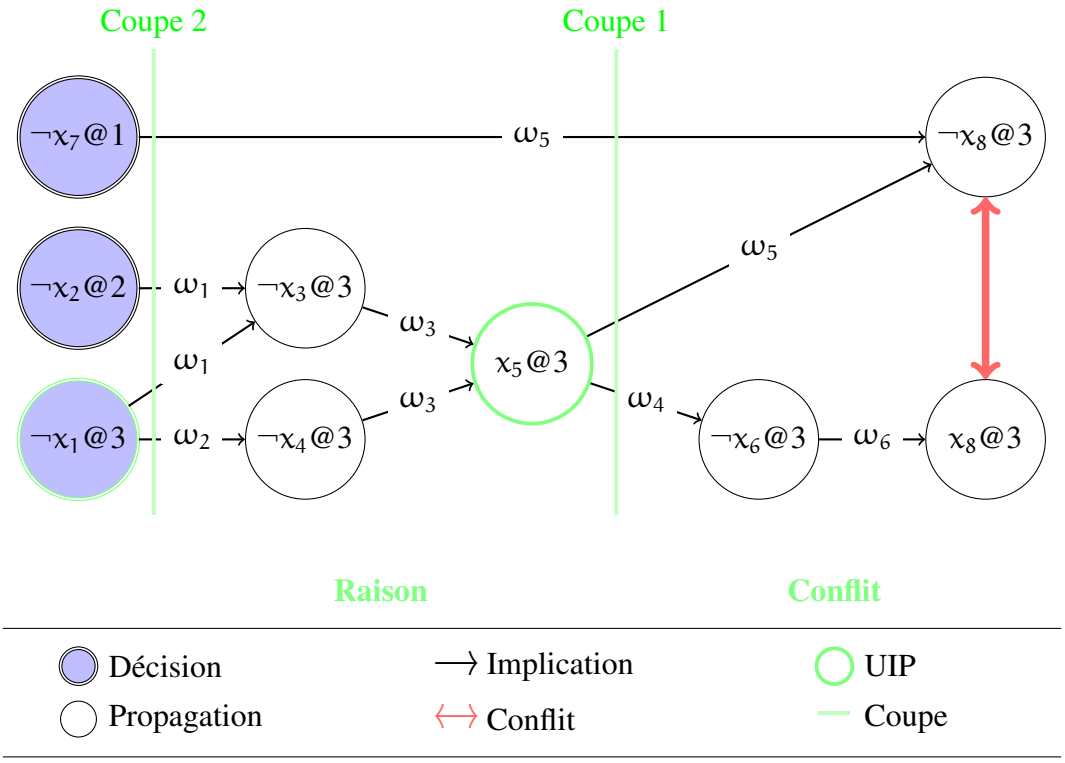


FIGURE 2.3 – Graphe d'implication

La combinaison de l'apprentissage des clauses de conflit et de ce mécanisme de backjump permet au solveur d'élaguer rapidement des zones conflictuelles de l'espace de recherche. Cependant, il arrive que le solveur se bloque dans la même zone de recherche pendant une longue période (il ne progresse pas de manière significative). Ce phénomène est connu sous le nom de *heavy tailing* [19]. Un moyen de sortir de cette zone est de relancer la recherche, tout en conservant les informations utiles (*e.g.,* clauses apprises). Techniquement, un redémarrage est un retour au niveau de décision 0, soit avant la première décision. La détection d'une zone de *heavy tailing* n'est pas un problème facile et a été largement traité dans la littérature [65, 8, 14]. Ces stratégies peuvent utiliser le nombre de conflits ou la profondeur (actuelle) de l'arbre de recherche comme heuristique pour décider de redémarrer.

2.2.5 *Heuristiques de branchement*

Nous avons rapidement présenté la procédure `choixLitteralBranchement` utilisée dans DPLL et CDCL, une procédure de décision, qui choisit le prochain littéral de *branchement* (appelé ainsi, car il permet de descendre plus profondément dans l'arbre de recherche). Pour être plus précis, deux décisions sont prises par cette procédure : le choix de la variable et le choix de la polarité de cette variable (fixer la variable à $\top$ ou $\bot$ dans l'affectation courante). (i) La première décision consiste à optimiser l'ordre de branchement des variables, la complexité de la résolution dépend fortement de l'ordre de branchement des variables et il est évidem-

ment impossible de trouver un ordre optimal de manière déterministe en temps polynomial. (ii) La deuxième décision consiste à fixer une valeur pour la variable qui a un sens selon les contraintes imposées par la formule, dans le cas des instances satisfaisables en particulier. Un exemple peut être que pour deux variables x et $y \in \varphi$, il n'existe aucun modèle de φ dans laquelle x et y ont la même valeur. Les contraintes les plus triviales peuvent être déduites lors de la propagation unitaire ou l'apprentissage de clause, mais il est évidemment difficile durant la résolution de découvrir la valeur "optimale" pour une variable donnée.

(i) Heuristiques pour optimiser l'ordre de branchement : La complexité de résolution des problèmes SAT est fortement reliée à la profondeur de l'arbre de recherche. Comme sa construction est dominée par le choix des variables de décision, le choix de *bonnes* variables de décision (*e.g.,* celles qui génèrent le plus de propagations unitaires) diminue considérablement le temps de calcul. De nombreuses heuristiques ont été définies pour augmenter la largeur de la propagation, réduisant ainsi la profondeur de l'arbre de recherche [25, 36, 40, 83, 53, 68]. Les solveurs modernes utilisent des heuristiques favorisant les variables impliquées dans les conflits les plus récents, à savoir, *VSIDS* [74] et *LRB* [61] :

— **Variable state independent decaying sum (VSIDS) [74]** est utilisée dans presque tous les solveurs SAT modernes. Pour chaque variable, un score est calculé durant la résolution : pour chaque conflit, les variables incluses dans les clauses traversées durant l'analyse du conflit sont récompensées. Quand une décision est nécessaire, la variable non affectée avec le score le plus élevé est choisie. La métrique VSIDS est très volatile et varie rapidement.

— **Learning rate branching (LRB) [61]** est une généralisation du VSIDS. Le processus de décision est vu comme un problème d'optimisation où l'objectif est de maximiser le taux d'apprentissage, défini comme la capacité des variables à générer des conflits et donc à apprendre des clauses. Cet algorithme est basé sur de l'apprentissage par renforcement et répond à un problème formalisé comme un *problème de bandit manchot*. Soit I l'intervalle de temps entre l'affectation d'une variable v et le moment où v n'est plus affectée. $L(I)$ est le nombre de clauses apprises pendant l'intervalle I. Soit $P(v, I)$ le nombre de clauses apprises dans lesquelles v a participé (c'est-à-dire que v est dans la clause ou v est dans le côté conflit du graphe d'implication (voir Section 2.2.3)) pendant l'intervalle I. Alors, le taux d'apprentissage est défini par $\frac{P(v,I)}{L(i)}$. Pour une variable donnée, les taux d'apprentissage les plus récents sont préférés aux plus anciens.

(ii) Heuristiques pour optimiser l'affectation des valeurs : Lorsqu'une variable de décision est choisie, sa *polarité* ($\top$ ou $\bot$) reste indéterminée. La stratégie mise en œuvre dans presque tous les solveurs est la *sauvegarde de la phase* (ou *progress saving*) [81]. Les instances industrielles contiennent des *composants*, comme le souligne [20]. Un composant est un sous-ensemble de clauses, et deux composants sont indépendants s'ils ne partagent aucune variable. Lorsqu'un conflit est atteint, la recherche peut sauter en arrière sur plusieurs niveaux. L'affectation de certaines variables est alors oubliée, ce qui peut conduire à la perte de la résolution de certains composants. Afin de contrer ce phénomène, la dernière affectation des

variables forcée par la propagation unitaire est stockée, dans une structure que nous appelons *la phase*. Lorsqu'une variable est choisie pour le prochain branchement, elle est affectée à la valeur stockée dans la phase. On peut résumer ce mécanisme comme un cache stockant les valeurs chaudes des variables.

2.2.6 *Nettoyer la base de clauses apprises*

Les clauses apprises ne sont pas toutes "utiles" et dans certains cas, leur présence peut introduire un coût supplémentaire à l'algorithme, notamment lors de la propagation unitaire. De plus, il n'est pas possible de conserver toutes ces clauses d'un point de vue matériel, en effet leur nombre peut croître exponentiellement avec le nombre de variables. Garder toutes ces clauses peut introduire une saturation de la mémoire et donc le ralentissement ou même l'arrêt du programme. Il faut définir une métrique permettant d'ordonner les clauses par leur utilité et supprimer les moins utiles. On peut noter que définir ces métriques peut être complexe, car elles entraînent forcément un biais dans l'étude. Par exemple, si l'on supprime fréquemment toutes les clauses de tailles supérieures à 10, forcément ces clauses auront moins de chance d'être utilisées par le solveur. Se basant sur des études empiriques, plusieurs articles de la littérature ont proposé des heuristiques :

— La *taille de la clause* est très souvent utilisée par les solveurs. En effet, une clause est vraiment utile lorsqu'elle participe à la propagation unitaire. Plus la clause est petite, plus sa chance de provoquer une propagation est grande.

— On peut également trier les clauses en réutilisant la logique utilisée pour définir des heuristiques de décision pour le branchement : notamment *l'activité* des clauses. On introduit un score qui détermine l'activité d'une clause. Quand une clause participe à l'analyse du conflit, son activité est incrémentée. Les clauses avec une activité faible sont supprimées.

— La *Literal Block Distance (LBD)* [7]. C'est une mesure qui calcule la qualité d'une clause selon le nombre de niveaux de décision présents dans la clause. Les clauses avec une haute valeur LBD sont supprimées.

2.2.7 *Preprocessing/Inprocessing*

Nous avons étudié dans cette section les algorithmes et heuristiques nécessaires à la construction d'un outil de résolution capable de résoudre une instance d'un problème SAT. Un composant essentiel de ces solveurs, appelé *preprocessing*, est activé avant même le début de la résolution. Nous pouvons distinguer deux types de preprocessing : les mécanismes qui s'appliquent directement à la formule pour la simplifier, et ceux qui extraient des informations de la formule afin d'initialiser les différentes métriques d'un solveur, telles que les heuristiques de décision. Dans les deux cas, si ces mécanismes peuvent reposer sur des algorithmes capables de dériver une solution pour la formule, ce n'est pas leur but premier. Des limitations

sont donc appliquées à ces mécanismes afin de limiter le temps de calcul. Nous décrivons dans cette section uniquement les mécanismes de preprocessing de la première catégorie. L'initialisation des solveurs sera abordée Chapitre 5.

2.2.7.1 *Simplification de la formule*

Dans ce cas, le but du preprocessing est de simplifier la formule d'entrée, le plus souvent en supprimant des variables et des clauses, bien que certaines techniques puissent également ajouter des contraintes (selon la même logique que l'apprentissage de clauses de conflit Section 2.2.3). L'objectif est d'éliminer toutes redondances ou complexité introduites lors de l'encodage du problème modélisé en une instance du problème SAT (nous rappelons que seule la forme CNF est considérée). Nous présenterons ici quelques-unes des techniques employées par les solveurs rencontrés au cours de cette thèse.

Bounded Variable Elimination [38] (BVE) : Cette technique implémentée dans le prépocesseur `SatElite` utilisé par des solveurs comme `MiniSat` et `MapleCOMSPS`, applique la *résolution* pour supprimer des variables. La *résolution* est à la base de l'algorithme **DP** [33], algorithme fondateur pour résoudre le problème SAT, qui précède l'algorithme **DPLL** (Section 2.2.1). Pour éliminer la variable x, *BVE* applique les étapes suivantes :

- La *résolution* : Pour toutes les paires de clauses $\omega_1 = (x \vee \ell_2^1, \ldots, \ell_n^1)$ et $\omega_2 = (\neg x \vee \ell_2^2, \ldots, \ell_m^2)$ sont générées les *clauses résolvantes* $\omega = (\ell_2^1, \ldots, \ell_n^1 \vee \ell_2^2, \ldots, \ell_m^2)$ privées de la variable x.
- Après avoir ajouté les *clauses résolvantes* à la formule, les clauses originales sont supprimées.
- Pour éviter la possible explosion mémoire (le problème fondamental de l'algorithme DP), BVE n'applique la réalisation sur une variable seulement si cela n'augmente pas le nombre de clauses. Dans les variantes modernes de cette technique, la limite sur l'augmentation autorisée de la taille de la formule est augmentée sur plusieurs tours de simplification si la taille n'a pas trop augmenté au tour précédent [75].

Élimination gaussienne [91] : Cette technique est axée sur la simplification d'une forme particulière appelée *contrainte XOR*. Pour un ensemble de variables donné, cette contrainte peut être représentée sous la forme $\oplus_i x_i = \top/\bot$. Une telle contrainte est fausse pour toute affectation qui affecte $\top$ à un nombre pair de littéraux. Pour tout ensemble de variables de taille n, s'il existe 2^{n-1} combinaisons différentes de négations dans la formule, une contrainte XOR peut-être extraite. Exemple :

$$\left.\begin{aligned} x_1 \vee x_2 \vee \neg x_3 &= \top \\ x_1 \vee \neg x_2 \vee x_3 &= \top \\ \neg x_1 \vee x_2 \vee x_3 &= \top \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 &= \top \end{aligned}\right\} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = \bot \tag{2.1}$$

$$\left.\begin{aligned} x_1 \vee x_2 \vee x_3 &= \top \\ \neg x_1 \vee x_2 \vee \neg x_3 &= \top \\ x_1 \vee \neg x_2 \vee \neg x_3 &= \top \\ \neg x_1 \vee \neg x_2 \vee x_3 &= \top \end{aligned}\right\} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = \top \tag{2.2}$$

Un ensemble de contraintes XOR peut alors être considéré comme un système d'équations linéaires pouvant être résolu par élimination gaussienne. Ce type de simplification s'est avéré particulièrement efficace pour des instances du problème SAT modélisant des problèmes du domaine de la cryptographie.

Vivification [80] : Cette technique vise à réduire la taille des clauses en supprimant les littéraux qui sont impliqués par le reste de la formule. Le problème de minimiser une clause en une *clause minimale* est lui-même NP-difficile. La vivification propose un algorithme incomplet, mais de complexité linéaire, permettant de détecter les redondances présentes dans les clauses de la formule. Pour ce faire, les auteurs de [80] se basent sur la propagation unitaire (Section 2.2.1). Nous définissons qu'une clause ω de φ est impliquée par la propagation unitaire si la propagation unitaire de la clause complémentaire $\overline{\omega}$ implique une affectation qui expose un conflit (voir Section 2.2.3). Pour une clause de n littéraux $\omega = \{\ell_1, \ldots, \ell_n\}$ l'algorithme applique la propagation unitaire sur les littéraux complémentaires de ω, donc $\{\neg\ell_1, \ldots, \neg\ell_n\}$. À l'issue de cette propagation, trois cas peuvent se produire :

— La propagation atteint un conflit : $\exists i < n$ t.q. $\varphi \backslash \{\omega\}|_{\{\neg\ell_1,\ldots,\neg\ell_i\}} \Rightarrow \bot$. Dans ce cas, l'algorithme ajoute la clause $\{\ell_1, \ldots, \ell_i\}$. Cette clause subsume l'originale, qui peut donc être supprimée.

— Autrement, une nouvelle clause est produite si un littéral non affecté de la clause voit sa valeur fixée par une propagation unitaire. Par exemple si la propagation unitaire implique un littéral négatif : $\exists i < j \leq n$ t.q. $\varphi \backslash \{\omega\}|_{\{\neg\ell_1,\ldots,\neg\ell_i\}} \Rightarrow \neg\ell_j$. La clause produite est $\{\ell_1, \ldots, \neg\ell_j\}$. Cette nouvelle clause et la clause ω originale permettent d'inférer la *clause résolvante* $\{\ell_1, \ldots, \ell_{j-1}, \ell_{j+1}, \ldots, \ell_n\}$. Cette clause subsume la clause originale, cette dernière peut donc être supprimée.

— De même, si le littéral positif est fixé : $\exists i < j \leq n$ t.q. $\varphi \backslash \{\omega\}|_{\{\neg\ell_1,\ldots,\neg\ell_i\}} \Rightarrow \ell_j$, la clause impliquée est $\{\ell_1, \ldots, \ell_j\}$, qui subsume la clause originale.

2.2.7.2 *Inprocessing*

À l'inverse du *preprocessing*, les mécanismes activés en cours de résolution sont appelés *inprocessing*. La distinction entre ces deux mécanismes repose principalement sur le moment de l'appel. Dans les faits, de nombreuses techniques d'inprocessing utilisent une version simplifiée des mécanismes utilisés pour le preprocessing. Leur avantage est qu'ils peuvent utiliser les informations découvertes par le solveur pour améliorer leur efficacité. Cependant, il est important de ne pas appeler ces mécanismes systématiquement afin de ne pas entraver la résolution.

Les algorithmes d'inprocessing définissent donc des seuils pour déterminer s'il est viable de les exécuter, tels que le nombre de variables affectées [27], le nombre de redémarrage depuis le dernier appel [27, 66] ou la taille des clauses considérées [66]. Des algorithmes d'inprocessing seront discutés plus en détail Chapitres 4 et 5. Dans ce dernier, nous présenterons nos travaux sur l'inprocessing.

2.3 RÉSOLUTION PARALLÈLE DU PROBLÈME SAT

Dans un passé récent, la puissance d'un processeur augmentait grâce à des fréquences d'horloges de plus en plus élevées. Outre les algorithmes et heuristiques abordés dans la section précédente, une manière de rendre la résolution plus rapide était simplement d'obtenir une machine avec un processeur plus rapide. De nos jours, les fabricants de puces électroniques sont en mesure de concevoir et de fabriquer à grande échelle des architectures composées de multiples cœurs de calcul, plus efficaces sur le plan énergétique. L'industrie s'est tournée vers ce type d'architecture, attirée par l'incroyable augmentation des performances possible en permettant à plusieurs applications de fonctionner en parallèle ou en répartissant les différentes tâches d'une application sur plusieurs unités de calcul.

Les développeurs d'applications ont dû réfléchir sérieusement à la conception de systèmes dont les composants peuvent être répartis sur plusieurs unités de calcul afin d'augmenter les performances sans introduire les bogues inhérents à ce paradigme de programmation. La standardisation de ces environnements multicœurs a également conduit à l'utilisation massive d'environnements distribués : qu'il s'agisse de grappes de machines connectées à un même réseau ou de larges grilles de calcul composées de machines souvent géographiquement éloignées les unes des autres. Ainsi, de nouvelles stratégies parallèles ont été développées permettant aux solveurs SAT d'exploiter cette puissance de calcul. L'objectif de cette section est d'étudier et de classer les différentes approches qui ont été développées dans le contexte de la résolution parallèle du problème SAT.

Il existe principalement deux familles de stratégies de parallélisation. La première regroupe les algorithmes de type *diviser pour régner* (nous utiliserons le terme *divide-and-conquer* par la suite) [101], où l'espace de recherche est divisé dynamiquement au cours de la résolution. Les sous-espaces de recherche résultants sont ensuite résolus par des algorithmes séquentiels standards. La deuxième famille regroupe les algorithmes de type *portfolio* [45]. Ces stratégies lanceront chaque travailleur sur l'ensemble de la formule, en veillant à ce que chaque travailleur traverse l'espace de recherche différemment, soit en utilisant un algorithme différent, soit en utilisant des paramètres différents pour le même algorithme. Ces deux approches sont présentées Section 2.3.1 et Section 2.3.2 respectivement.

Dans ce contexte parallèle, nous pourrons appeler un algorithme séquentiel exécuté par un fil d'exécution un *travailleur* et un sous-espace de recherche (dans le cas du divide-and-conquer) une *tâche*. Dans les deux approches, les algorithmes séquentiels sous-jacents doivent partager dynamiquement les informations apprises (principalement, les clauses apprises). De nombreuses heuristiques tentent d'améliorer ce partage en proposant un compromis entre les gains

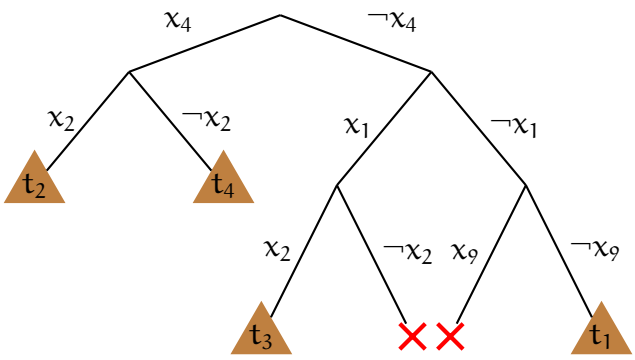


FIGURE 2.4 – Utilisation du guiding path pour diviser l'espace de recherche

qu'il apporte et son coût matériel et logique. Ainsi, une stratégie de partage qui distribue les clauses apprises entre les différents travailleurs doit paramétrer le débit d'envoi en fonction des contraintes de mémoire imposées par la machine sur laquelle elle s'exécute. Mais dans un même temps, elle doit veiller à ne pas ralentir la propagation unitaire des différents travailleurs par des clauses qui leur seront inutiles par rapport à leurs positions dans l'espace de recherche. Une mise en œuvre réfléchie de ce partage est importante et sera examinée plus en détail Section 2.3.3.

2.3.1 *Divide-and-Conquer*

La stratégie divide-and-conquer est basée sur la division de l'espace de recherche en sous-espaces qui sont ensuite résolus par différents travailleurs. S'il est prouvé qu'un sous-espace est SAT alors la formule initiale est SAT. La formule est UNSAT si tous les sous-espaces sont UNSAT. Les difficultés introduites par ce paradigme sont présentées dans les sous-sections suivantes : la division de l'espace de recherche (Section 2.3.1.1) ; la conception d'une heuristique pour équilibrer la complexité des sous-espaces générés (Section 2.3.1.2) ; la mise en place politique de répartition dynamique des tâches (Section 2.3.1.3) et le partage des clauses apprises (Section 2.3.1.4).

2.3.1.1 *Techniques pour diviser l'espace de recherche*

Pour diviser l'espace de recherche, la technique la plus utilisée est le *guiding path* [101]. Nous présenterons également dans cette section le *scattering* [52] et le *partitionnement xor* [82].

Guiding Path. Un solveur va traditionnellement commencer sa traversée de l'arbre de recherche depuis la racine. Un guiding path est simplement un chemin imposé au solveur pour commencer sa recherche. On peut le voir comme une tentative de résoudre la formule en fonction d'un ensemble d'hypothèses sur certaines variables, hypothèses sur lesquelles le solveur ne revient jamais (les autres hypothèses possibles seront traitées par une autre instance). Il

prend la forme d'un cube (pour rappel une conjonction de littéraux) qui est assumé par le travailleur. Soit la formule φ et la variable x. Grâce à la décomposition de Shannon [86], nous pouvons réécrire φ comme $\varphi = (\varphi \wedge x) \vee (\varphi \wedge \neg x)$. Les deux guiding paths sont réduits à un littéral : (x) et $(\neg x)$. Ce principe peut être appliqué de manière récursive sur chaque sous-espace pour créer des chemins plus longs.

Figure 2.4 présente cette approche dans un exemple où six sous-espaces sont créés depuis la formule originale. Ils sont issus des guiding paths suivants : $(x_4 \wedge x_2)$, $(x_4 \wedge \neg x_2)$, $(\neg x_4 \wedge x_1 \wedge x_2)$, $(\neg x_4 \wedge x_1 \wedge \neg x_2)$, $(\neg x_4 \wedge \neg x_1 \wedge x_9)$, $(\neg x_4 \wedge \neg x_1 \wedge \neg x_9)$. Les sous-espaces prouvés UNSAT sont mis en valeur par une croix rouge. Les sous-espaces restants sont soumis aux travailleurs (notés t_i).

Scattering (ou *dispersion*). Contrairement au guiding path, cette approche introduit de nouvelles contraintes à la formule pour créer des sous-formules qui vont contraindre le solveur dans une partition du problème original. Le fonctionnement est le suivant, pour une formule φ donnée et n le nombre de partitions souhaitées (aussi appelé *scattering factor* soit *facteur de dispersion*), la construction de la sous-formule φ_i est donnée par l'équation suivante :

$$\varphi_i = \begin{cases} \varphi \wedge \mathcal{G}_1, \text{ if } i = 1 \\ \varphi \wedge \neg\mathcal{G}_1 \wedge \ldots \wedge \mathcal{G}_{i-1} \wedge \mathcal{G}_i, \text{ if } 1 < i < n \\ \varphi \wedge \neg\mathcal{G}_1 \wedge \ldots \wedge \neg\mathcal{G}_n, \text{ if } i = n \end{cases}$$

Où $\mathcal{G}_i$ est un cube de d_i littéraux de φ : $\mathcal{G}_i = (\ell_1 \wedge \ldots \wedge \ell_{d_i})$. La négation du cube $\mathcal{G}_i$ est la clause $\neg\mathcal{G}_i = (\ell_1 \vee \ldots \vee \ell_{d_i})$. Le nombre d_i est choisi tel que φ est divisé en n partition de taille égale.

Partitionnement Xor. Le *partitionnement Xor* s'applique à une formule φ sur n littéraux pour générer les deux sous-formules : $\varphi_{pair} = \varphi \wedge (\ell_1 \oplus \ldots \oplus \ell_n \oplus 1)$ et $\varphi_{impair} = \varphi \wedge (\ell_1 \oplus \ldots \oplus \ell_n \oplus 0)$. Durant la résolution de φ_{pair} (φ_{impair}), un nombre pair (impair) parmi les n littéraux doit être affecté à vrai pour satisfaire *la contrainte xor*. Si d'autres partitions sont nécessaires, nous pouvons simplement réappliquer le partitionnement Xor sur les partitions générées. Cette technique présente l'avantage de générer des sous-espaces avec une répartition équilibrée des modèles dans le cas d'une formule SAT. On peut noter qu'utiliser ce mécanisme avec $n = 1$ fournira un résultat équivalent au guiding path.

2.3.1.2 *Choix d'une variable de décision*

Quelle que soit la technique de division de l'espace de recherche choisie (Section 2.3.1.1), la qualité de leur implémentation dépendra de la sélection des variables ou littéraux utilisée pour effectuer cette division. Choisir la meilleure *variable de décision* est un problème difficile, nécessitant la création d'une heuristique adaptée. La mesure de qualité d'une bonne *heuristique de division* est sa capacité à créer des sous-espaces équilibrés en termes de temps requis

pour les résoudre. En effet, des sous-espaces trop simples à résoudre amènent le travailleur à constamment demander de nouvelle tâche et donc rediviser l'espace de recherche. Ce phénomène (connu comme *l'effet ping-pong*) conduit à une situation où autant, sinon plus, de temps de calcul est consacré à cette communication entre le travailleur et l'entité chargée de diviser l'espace de recherche, qu'au temps de résolution. De plus, elle doit évidemment améliorer le temps de résolution globale comparée à une résolution séquentielle.

Les heuristiques de division peuvent être classées en deux catégories : *look-ahead* et *look-back*. Une heuristique de type look-ahead se calibre en essayant de prédire le comportement futur du solveur. Au contraire, les heuristiques de type look-back sont calculées à partir de statistiques accumulées durant la résolution. Nous présentons ici certaines de ces heuristiques les plus importantes.

Look-Ahead. Pour prédire le comportement futur du solveur, les algorithmes look-ahead utilisent un solveur DPLL (Algorithme 1) pour évaluer à l'avance la qualité d'une branche de l'arbre de recherche [47]. Tout d'abord, un ensemble de variables est sélectionné en fonction de diverses heuristiques. Le solveur DPLL applique la propagation unitaire sur une des variables de l'ensemble. Si aucun conflit n'est dérivé, il mesure la différence entre la formule originale et la formule simplifiée par la propagation. Il va ensuite revenir sur ce choix (annuler les simplifications induites par la propagation) et réappliquer la propagation unitaire sur une autre variable de l'ensemble. La variable qui aura généré la sous-formule la plus intéressante, selon la mesure de différence choisie par l'algorithme, est sélectionnée pour la division. Par exemple, la méthode appelée *cube-and-conquer* [49] utilise une heuristique de mesure recherchant les variables impliquant le plus grand nombre de propagations unitaires, afin de créer des sous-espaces de recherche le plus petit possible (*i.e.,* avec le moins de variables non affectées). Le problème principal de cette technique est le coût généré par l'application de la propagation unitaire pour l'exploration des différentes variables candidates.

Look-Back. Inversement, les algorithmes de type look-back guident le solveur sur la base de son comportement passé. Nous décrivons maintenant les heuristiques de type loop-back utilisées pour sélectionner des variables de décision :

— Puisque les solveurs séquentiels se basent sur des heuristiques pour sélectionner leurs variables de décision, celles-ci peuvent naturellement être utilisées pour opérer la division de l'espace de recherche. Une possibilité est d'utiliser *l'ordre de branchement* des variables imposées par l'heuristique VSIDS (voir Section 2.2.5) pour décomposer la recherche en sous-espaces. En effet, il est communément admis qu'une variable avec à un haut rang dans cet ordre constitue un bon point de départ pour un nouvel espace d'exploration [52, 70, 6].

— Le nombre de *flips (ou renversement de valeur)* des variables [4], c'est-à-dire le nombre de fois qu'une propagation unitaire affecte une variable à l'inverse de sa dernière valeur propagée. Par conséquent, ordonner les variables en fonction du nombre de leurs flips et choisir la plus élevée comme point de division permet de générer des sous-espaces

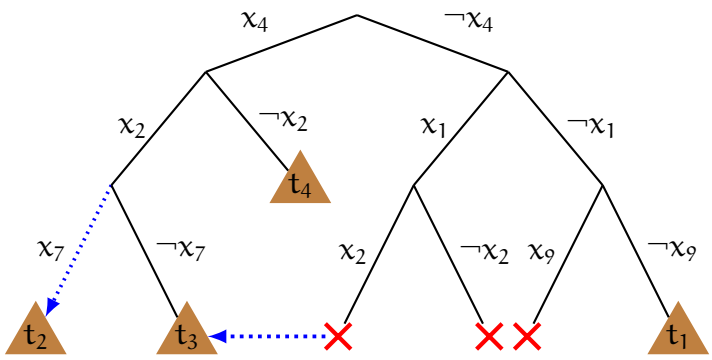


FIGURE 2.5 – Équilibrage des charges dynamique utilisant le vol de travail

de recherche avec un temps de calcul comparable. Les auteurs de [4] l'utilisent pour diminuer le coût de l'heuristique look-ahead en présélectionnant un pourcentage défini de variables ayant le plus grand nombre de flips. Ceci permet de limiter le nombre de variables sur lesquelles la propagation est appliquée.

— Nous présentons une dernière approche appelée *taux de propagation* [77]. Elle tend à produire le même effet que l'heuristique look-ahead (présentée ci-dessus), mais en analysant des statistiques accumulées durant la résolution. Le taux de propagation d'une variable x est le ratio entre le nombre de propagations dues à un branchement sur x et le nombre de fois où x a été choisie comme décision. La variable ayant le taux le plus élevé est choisie comme point de division.

2.3.1.3 *Équilibrage dynamique des charges*

Malgré tous les efforts déployés dans la conception d'heuristiques de division permettant d'équilibrer les sous-espaces générés, il est en pratique impossible d'assurer un équilibre optimal entre les partitions. Ainsi, certains travailleurs deviennent rapidement oisifs. De ce constat découle la nécessité de concevoir un mécanisme d'équilibrage des charges dynamique. Il existe essentiellement deux stratégies : *le vol de travail* et *la file d'attente*.

Vol de travail. Une première solution pour réaliser cet équilibrage consiste à rediviser des sous-espaces prouvés plus complexes afin de créer de nouvelles tâches pour les travailleurs oisifs. À chaque fois qu'un travailleur prouve que son sous-espace est UNSAT, il demande une nouvelle tâche. Un travailleur cible est choisi pour diviser son espace de recherche, en utilisant un des mécanismes de division présentés. Ainsi, le travailleur cible est affecté à l'un des nouveaux sous-espaces générés, tandis que le solveur va commencer à travailler sur l'autre.

Lorsqu'une nouvelle division est nécessaire, le choix de la meilleure cible est un problème difficile. Par exemple, dans `Dolius` [4], la stratégie consiste à utiliser un ordre FIFO pour élire les cibles : le prochain travailleur ciblé par le vol de travail est celui qui travaille depuis le plus longtemps sur son espace de recherche. Cette stratégie garantit l'équité entre les travailleurs.

Elle a un autre avantage dans le cas où l'heuristique pour choisir une variable de division est de type look-back, le travailleur cible ayant une bonne connaissance de son espace de recherche.

Considérons l'exemple présenté dans la Figure 2.4. Supposons que le travailleur t_3 prouve son sous-espace comme étant UNSAT et demande donc une nouvelle tâche. Le travailleur t_2 est choisi pour diviser et partager son sous-espace. Dans la Figure 2.5, x_7 est choisie comme variable de division et deux nouveaux *guiding path* sont créés, $(x_4 \wedge x_2 \wedge x_7)$ pour t_2 et $(x_4 \wedge x_2 \wedge \neg x_7)$ pour t_3.

File d'attente. Une autre solution pour éviter la latence entraînée par des travailleurs en attente est de créer avant le démarrage de la résolution plus d'espaces de recherche que de travailleurs disponibles (cube-and-conquer [49]). Un solveur look-ahead est utilisé pour générer un grand nombre de tâches. Elles sont stockées dans une file d'attente où les travailleurs en besoin de travail pourront les récupérer. Pour augmenter le nombre de tâches disponibles dynamiquement, un travailleur cible est choisi pour diviser son espace de recherche. Une variante de cette stratégie implémentée dans `Treengeling` [16] choisie la tâche avec le plus petit nombre de variables pour la division, ce qui a tendance à favoriser les instances SAT. Lorsque la file d'attente est vide, la stratégie peut toujours basculer sur le vol de travail.

2.3.1.4 *Partage des clauses apprises*

Le partage de clauses apprises dans un solveur parallèle de type divide-and-conquer peut être limité selon la méthode de division choisie. Diviser un espace de recherche consiste à définir des contraintes sur les valeurs de certaines variables. Comme vu précédemment, il existe deux techniques pour implémenter ces contraintes : (i) contraindre la formule originale en y ajoutant de nouvelles clauses ; (ii) contraindre la routine de décision du solveur.

Lorsque la division est réalisée avec (i), certaines clauses apprises ne peuvent pas être partagées entre les travailleurs. C'est le cas des clauses apprises déduites depuis au moins une des clauses ajoutées pour la division. La solution la plus simple pour conserver l'exactitude de la résolution est de désactiver le partage de clause [16]. Une autre approche est de marquer les clauses qui ne doivent pas être partagées [55]. Les clauses ajoutées pour la division sont marquées et si ces clauses participent à la déduction de nouvelles clauses leurs *tags* sont propagés à ces dernières.

Si au contraire la division est réalisée avec (ii), il n'y a pas besoin de restrictions pour le partage de clauses apprises. Cette solution est souvent implémentée en utilisant un mécanisme de *supposition* (comme le guiding path par exemple), c'est à dire forcer la valeur de certaines variables au début de la résolution du solveur. On peut trouver cette solution implémentée dans les solveurs `Dolius` [4] ou `AmPharoS` [6].

2.3.2 *Portfolio*

Le paradigme portfolio est introduit avec le solveur `ManySat` [45] et domine encore aujourd'hui le domaine des solveurs SAT parallèles. Cette affirmation est basée sur les excellentes performances des différents solveurs de ce type dans les dernières compétitions SAT. Dans un portfolio, tous les travailleurs s'attèlent à résoudre l'intégralité de la formule, le premier à trouver une solution met fin au calcul. Deux concepts sont importants pour comprendre le succès de cette technique : la *diversification* et l' *intensification*.

2.3.2.1 *Diversification*

La diversification consiste à faire varier le comportement des différents travailleurs. Le but est de visiter l'espace de recherche de manière significativement différente pour augmenter les chances de terminer rapidement la résolution. De multiples stratégies existent pour appliquer la diversification. Elles peuvent être tout aussi bien utilisées individuellement qu'en combinaison. Parmi lesquelles on peut citer : la **Paramétrisation**, la **Composition d'heuristiques**, **la Division souple**, l'**Intention** et le **Branchement par bloc**.

Paramétrisation. Une manière simple de garantir la diversification est d'utiliser le même algorithme séquentiel pour chaque travailleur, mais avec ses paramètres initialisés avec différentes valeurs. Ces paramètres peuvent être des graines aléatoires, des valeurs différentes pour des heuristiques données (comme des seuils d'application par exemple).

Composition d'heuristiques. Puisqu'il existe de multiples heuristiques dans le contexte séquentiel, on peut instancier des solveurs avec des heuristiques différentes pour les différentes routines principales d'un solveur séquentiel (présentées le long de la Section 2.1). Leurs différences peuvent porter sur la stratégie de décision, de redémarrage, le schéma d'apprentissage, etc. Par exemple, dans `ManySat` [45] plusieurs travailleurs sont instanciés avec des différences sur leurs stratégies de redémarrage, heuristiques de décision, et méthodes d'apprentissage. Il est également possible d'utiliser des solveurs séquentiels complètement différents en même temps, comme dans `HordeSat` [11].

Division souple. Une autre stratégie pour assurer la diversification s'appuie sur des travailleurs utilisant la sauvegarde de la phase (voir Section 2.2.5). Dans `HordeSat`, avant de commencer la recherche, chaque solveur reçoit une phase prédéfinie. Celle-ci agit en fait comme une division souple de l'espace de recherche. En effet, si la phase agit fortement sur l'espace de recherche visité initialement par un solveur, contrairement aux *suppositions* de variables, elle peut être changée au cours de la résolution par la routine de propagation unitaire.

Intention. Certains travaux ont proposé d'orienter les comportements d'un solveur parallèle en fonction des *intentions* de ses travailleurs [43]. L'intention d'un travailleur est simplement sa *phase* (voir Section 2.2.5) à un moment donné de la résolution. Une similarité d'intention est calculée pour chaque paire de travailleurs en utilisant une distance de Hamming sur leurs phases respectives. Si deux solveurs ont des intentions similaires, l'un d'eux verra sa phase inversée. Dans [43], cette idée est implémentée au sein d'un solveur `MiniSat`. Les intentions entre paires de travailleurs sont calculées tous les 5000 conflits. Deux solveurs ont une intention similaire si la distance de Hamming entre leurs phases est inférieure ou égale à 0,1.

Branchement par bloc (*Block branching*). Une troisième technique pour assurer la diversification est le *branchement par bloc* [89]. Chaque travailleur se concentre sur un sous-ensemble (ou bloc) particulier de variables. Pour chaque travailleur, les ordres de branchement des variables du bloc sont périodiquement modifiés. Par exemple, dans [88], l'ordre de branchement est modifié en augmentant fortement la valeur VSIDS des variables du bloc après un nombre fixe de redémarrages. Cette stratégie oblige les travailleurs à choisir d'abord les variables de décision dans leur propre sous-ensemble. Une façon d'obtenir ces sous-ensembles est d'analyser la relation entre les variables en utilisant une structure d'Union-Find comme décrit dans [89]. Les variables sont ensuite fusionnées en un bloc utilisant les clauses binaires présentes dans la formule initiale. Une autre idée consiste à utiliser la structure de communauté de la formule représentée sous forme de graphe [90] (le concept de communautés est discuté plus loin, Chapitre 3).

2.3.2.2 *Intensification*

Le concept d'intensification a été introduit dans [42] : certains travailleurs réexplorent différemment des régions de l'espace de recherche qui ont déjà été (partiellement) explorées par un autre travailleur (appelé *maître* dans ce paradigme). Cela permet d'*intensifier* la connaissance globale de cette partie de l'espace de recherche. À partir de l'exploration initiale d'un espace de recherche particulier, plusieurs informations sont connues : l'ensemble des littéraux de décision $\mathcal{D}$ (*i.e.,* les décisions prises et leurs polarités) et Λ, l'ensemble des clauses apprises pendant la recherche. L'ensemble Λ est ajouté au solveur pour garantir qu'il ne fera pas le même travail. De plus, cette information est utilisée pour construire un ensemble de littéraux de décision possibles pour la nouvelle traversée.

Les auteurs de [42] ont proposé trois approches. Dans la première, l'intensification prend exactement les mêmes décisions que le maître (*i.e.,* $\mathcal{D}$). Dans la deuxième technique, elle utilise comme littéraux de décision l'ensemble d'assertions (*i.e.,* la négation du 1er-UIP du graphe d'implication qui a conduit aux clauses apprises dans Λ). La troisième solution utilise comme littéraux de décision l'ensemble de conflits (*i.e.,* les littéraux incluent entre le 1er-UIP et la variable de conflit du graphe d'implication qui a conduit à l'apprentissage d'une clause dans Λ). La troisième technique semble être la plus efficace selon leurs résultats expérimentaux. Lorsque le maître redémarre, les travailleurs redémarrent également et entrent dans une nouvelle phase d'intensification.

La diversification et l'intensification sont deux axes orthogonaux. Si les solveurs impliqués dans le processus de diversification sont appelés des maîtres et que ceux qui sont en en charge de l'intensification sont considérés comme des travailleurs, les auteurs de [42] nous donnent la meilleure configuration selon leurs expériences : chaque maître doit être associé à un travailleur dédié à l'intensification.

2.3.3 *Partage*

Dans le contexte parallèle, un travailleur peut se trouver dans le même sous-espace de recherche qu'un autre travailleur. Même dans le cas du divide-and-conquer, il est possible de trouver des sous-arbres identiques malgré des cubes de division différents. Le partage de clauses apprises est une composante importante pour éviter ce problème et constitue le cœur de notre thèse.

L'information la plus importante qui peut être partagée est celle apportée par les clauses apprises par chaque travailleur lors de la résolution. Avec les deux paradigmes classiques, divide-and-conquer et portfolio, mais aussi avec l'utilisation d'approches hybrides, le partage de clauses apprises est possible. En pratique, toutes les clauses apprises ne peuvent pas être partagées entre tous les travailleurs, en raison des limitations matérielles, mais aussi pour éviter de ralentir le déroulement de l'algorithme en augmentant le coût de la propagation unitaire. La bonne conception d'une politique de partage demande de se poser les questions suivantes : *Quelles clauses doivent être partagées ? Et entre quels travailleurs ?* La première question implique comme dit précédemment que certaines clauses doivent être filtrées. De plus, il existe trois moments où ce filtre peut être mis en œuvre : lorsque les clauses sont exportées ; lorsque les clauses sont en transit pendant la phase de partage ; et enfin lorsque les clauses sont effectivement importées. Quant à la deuxième question, une réponse simple adoptée dans presque tous les solveurs SAT parallèles, consiste à partager les clauses entre tous les travailleurs. Cependant, une solution plus fine (mais plus complexe) consiste à laisser chaque travailleur choisir ses émetteurs.

2.3.3.1 *Export de clauses*

Comme vu dans la Section 2.2.6, la notion de clause utile est déjà discutée dans le contexte séquentiel afin d'optimiser le nettoyage de la base de clauses apprises. Nous pouvons réutiliser les mesures définies pour établir l'utilité d'une clause (*i.e.,* activité, taille, LBD [7]), dans le contexte parallèle. Dans de nombreux solveurs, seules les clauses inférieures à un certain seuil fixe pour l'une de ces mesures sont partagées. Par exemple, dans `ManySat` [45], seules les clauses de taille inférieure ou égale à 8 sont partagées.

S'il fait sens de réutiliser ces heuristiques dans la conception d'une stratégie de partage, les seuils existants sont calibrés pour répondre aux besoins en performance de l'algorithme séquentiel. Le partage de clauses ajoutant des contraintes sur la bande passante du bus mémoire,

la communauté a proposé de rendre ces seuils dynamiques au cours de la résolution. Cela permet un contrôle fin du flux des clauses apprises pendant le temps de résolution. En se basant sur l'algorithme *additive increase, multiplicative decrease* (AIMD) utilisé dans TCP pour éviter la congestion, les auteurs de [44] ont proposé d'ajuster dynamiquement la taille des clauses échangées entre paires de travailleurs. Dans `HordeSat` [11], le partage est limité à chaque tour à un certain nombre de littéraux (*i.e.,* somme de la taille des clauses).

Enfin, malgré les efforts de diversification de la recherche, il est tout à fait possible que deux travailleurs différents apprennent et donc partagent la même clause. Comme des mécanismes de détection de ces doublons peuvent être coûteux en mémoire (l'utilisation d'une simple map nécessite de stocker des dizaines de millions de hashs par exemple), de nombreux solveurs [11, 85] se tournent vers des structures d'ensemble probabiliste comme le filtre de Bloom [22]. Cette structure est plus légère en mémoire et permet de déterminer avec certitude qu'une clause n'a jamais été partagée. La contrepartie est qu'elle peut produire des faux positifs, c'est-à-dire qu'elle peut considérer une clause comme un doublon même si elle ne l'est pas.

2.3.3.2 *Import de clauses*

La LBD d'une clause (Section 2.2.6) ou l'activité des variables sont des mesures locales à un travailleur. Une clause utile, sur la base de ces mesures, peut être non pertinente dans le contexte d'un autre travailleur. Lorsqu'un travailleur reçoit des clauses, il doit pouvoir vérifier si ces clauses seront utiles compte tenu de sa position dans l'espace de recherche. Afin de calculer la pertinence des clauses reçues, des mesures telles que celle basée sur la sauvegarde de la phase appelée *Progress Saving based quality Measure(PSM)* [5] peuvent être utilisées. La PSM d'une clause est la cardinalité de l'intersection entre la clause (vue comme un ensemble de littéraux) et la phase du travailleur. Une valeur élevée est défavorable, car elle signifie que les clauses seront trop rapidement satisfaites par l'exploration courante. De plus, les auteurs de [5] proposent de ne pas supprimer les clauses avec un PSM élevé, au lieu de cela ces clauses sont *gelées* et peuvent être incorporées plus tard dans la base de données du solveur si leur PSM a diminué. Si ce n'est pas le cas, elles peuvent être définitivement supprimées. Une clause *gelée* n'est pas surveillée par le solveur, et n'est donc utilisée ni pour la propagation unitaire ni pour la détection de conflit.

Dans `Syrup` [10], lorsqu'une clause est importée, elle est ajoutée à une structure appelée *1-watched*, où seul un de ses littéraux est observé. Cette structure permet la détection de conflit (lorsqu'une valeur conflictuelle est attribuée au littéral observé), mais les clauses ne peuvent pas être utilisées pour la propagation unitaire. Lorsqu'une clause 1-watched déclenche un conflit, elle est promue au mécanisme de gestion de clauses classique. L'idée ici est de ne pas surcharger la propagation unitaire avec des clauses importées et de n'incorporer que les nouvelles clauses qui sont pertinentes (celles qui ont déjà provoqué un conflit). Cette base de données utilisant la structure 1-watched est souvent appelée *purgatoire*.

Les clauses peuvent être gérées à l'aide d'un mécanisme de promotion avec différents états. Par exemple, dans le solveur `AmPharoS` [6], il existe trois états : `en veille`, `purgatoire`,

et `apprises` (*i.e.,* la base de données de clauses apprise classique du solveur séquentiel). Lorsqu'une clause est reçue, elle est placée dans l'état `en veille` : la clause n'est pas encore attachée au solveur séquentiel. Les clauses dans l'état `purgatoire` ou dans l'état `apprises` sont attachées au solveur. Périodiquement, un certain nombre de clauses sont promues de `en veille` à `purgatoire`, selon une mesure combinant la PSM et la mesure *redundancy shared clauses measure (RSCM)*. Cette dernière indique le ratio entre le nombre de clauses reçues et le nombre de clauses conservées dans une fenêtre de temps glissante. Une clause n'est pas conservée si elle peut être subsumée par une autre clause déjà présente. En somme, plus les clauses reçues sont conservées, et donc plus la rscm est grande, moins nous voulons les promouvoir pour éviter de surcharger le solveur (et vice versa). Une clause est promue de l'état `purgatoire` à l'état `apprises` lorsqu'elle a été utilisée au moins une fois dans une analyse de conflit. Si une clause n'est pas promue après plusieurs analyses, elle est supprimée. Ce type de stratégie évite que le mécanisme de propagation unitaire des solveurs sous-jacents soit inondé par le nombre de clauses apprises.

2.3.4 *`Painless` : Un framework pour le développement et l'évaluation de solveurs SAT parallèles*

Nous avons présenté plusieurs stratégies pour paralléliser la résolution du problème SAT. En pratique, leurs performances dépendent largement de leur implémentation. Il est difficile de comparer équitablement différentes stratégies de parallélisation ou de partage. Il est possible de comparer les performances de solveurs parallèles entiers en les exécutant sur la même machine. Cependant, comparer la qualité des blocs individuels est plus complexe. Le framework `Painless` [56, 57] a été développé pour faciliter l'implémentation et la comparaison équitable de différentes stratégies. C'est un framework modulaire, écrit en C++, qui permet l'évaluation de différents algorithmes de résolution, de stratégies de distribution du travail et de politiques de partage à l'aide d'une interface commune. Grâce à `Painless`, les développeurs de solveurs SAT séquentiels peuvent exécuter leurs algorithmes sur plusieurs threads sans se soucier des défis techniques liés à la programmation concurrente. De plus, les développeurs intéressés par l'élaboration de politiques de parallélisation et de stratégies de partage d'informations peuvent le faire sans avoir à développer leur propre solveur SAT séquentiel. `Painless` est une application multithreadée, donc son démarrage ne crée qu'un seul processus. Ensuite, l'application crée plusieurs threads pour chaque travailleur ainsi que pour la gestion du partage. Ainsi, les différents threads de l'application communiquent grâce au *tas* du processus, partagé entre les threads.

Intégration des moteurs séquentiels Pour une intégration transparente de différents moteurs séquentiels, il est essentiel d'encapsuler les solveurs dans une interface de programmation d'application (API) unique. L'interface `SolverInterface` agit comme un adaptateur pour les fonctions de base attendues d'un solveur séquentiel et se divise en deux catégories : *distribution du travail* et *distribution des clauses apprises*.

Distribution du travail :

— `getDivisionVariable()` : renvoie une variable permettant de diviser l'espace de recherche, particulièrement utile dans le cas d'un divide-and-conquer.

— `diversify(int id)` : fixe les heuristiques d'un travailleur, particulièrement utile dans le cas d'un portfolio (Section 2.3.2). `id` identifie un travailleur.

— `solve(vector<lit> guiding_path)` : lance la résolution avec le vecteur de littéraux `guiding_path` comme hypothèse de départ s'il n'est pas vide.

— `setPhase(int var, bool val)` : change la valeur d'une variable fournie par l'heuristique de décision (Section 2.2.5).

— `addClauses(vector<Clause> clauses)` : ajoute des clauses de manière permanente à la formule. Ces clauses ne peuvent pas être supprimées contrairement aux clauses apprises importées. C'est utile dans le cas de stratégies guidant le solveur avec de nouvelles contraintes.

Distribution des clauses apprises :

— `addLearnedClauses(vector<Clause> clauses)`, `getLearnedClauses()` : La première méthode exporte les clauses apprises d'un travailleur, la seconde importe les clauses apprises provenant d'autres travailleurs. La manière dont ces clauses sont exportées/importées est à la charge du développeur de l'algorithme séquentiel. Nous verrons plus loin comment `Painless` gère traditionnellement cet aspect.

— `increaseClauseProduction()`/`decreaseClauseProduction()` : Demande au travailleur d'exporter plus/moins de clauses. La stratégie de partage fait appel à cette méthode, mais son implémentation est à intégrer dans le moteur séquentiel.

Dans `Painless`, la définition du filtre déterminant quelles clauses doivent être partagées se trouve dans l'implémentation de la `SolverInterface`. En effet, le solveur est plus à même d'avoir accès aux heuristiques permettant de déterminer l'intérêt d'une clause. Dans la plupart des implémentations de `Painless`, la distribution des clauses apprises se fait de manière asynchrone. Chaque implémentation de `SolverInterface` contient deux *tampons* de clauses, un *tampon d'import* et un *tampon d'export*. Ainsi, la méthode `addLearnedClauses()` place les clauses apprises dans le *tampon d'export* et `getLearnedClauses()` récupère les clauses venant des autres solveurs depuis le *tampon d'import*. Ces tampons sont implémentés sous la forme de liste *lockfree* [73]. Nous verrons plus loin le composant chargé de distribuer les clauses apprises entre les divers tampons.

Pour intégrer un nouveau solveur séquentiel dans `Painless`, il faut créer une nouvelle classe qui implémente une interface appelée `SolverInterface`. Cette classe implémente les méthodes requises, en enveloppant efficacement les méthodes fournies par l'API du solveur sous-jacent. Plusieurs adaptateurs ont été développés par les auteurs de `Painless` [57] et la communauté. Des adaptateurs sont disponibles pour des solveurs séquentiels populaires tels que `Kissat` [18, 28], `MapleCOMSPS` [62] ou `Lingeling` [15].

Composant de gestion de la parallélisation. La Section 2.3 présente les stratégies de parallélisation de base, telles que le portfolio et le divide-and-conquer. Painless permet l'intégration

de nouvelles stratégies ainsi que la combinaison de stratégies grâce à un mécanisme de combinaison structurée en arbre avec une profondeur arbitraire. Les nœuds internes représentent les stratégies de parallélisation et les feuilles représentent l'abstraction des threads exécutant les algorithmes séquentiels.

Dans le framework `Painless`, les nœuds sont créés en implémentant l'interface `WorkingStrategy`, tandis que les feuilles sont représentées par une instance de la classe `SequentialWorker`, qui est une sous-classe de `WorkingStrategy` intégrant un flux d'exécution (un thread) exécutant son solveur séquentiel associé. Le flux de résolution au sein de cette structure arborescente est défini par l'utilisateur et contrôlé par les deux principales méthodes de WorkingStrategy :

— `solve(vector<lit> guiding_path)` : propage l'ordre de débuter la résolution. Cette propagation se fait de la racine vers les feuilles.

— `join()` : propage les résultats d'une feuille vers la racine.

En plus de ces méthodes, `WorkingStrategy` exige l'implémentation de méthodes avec la même signature que les méthodes de distribution du travail de la `SolverInterface` présentées ci-dessus. Le développeur pourra implémenter ces méthodes et éventuellement modifier la logique du thread `SequentialWorker` pour créer sa stratégie.

Il existe une version de `Painless` intégrant une stratégie portfolio [57, 94] et divide-and-conquer [58].

Composant de gestion du partage des clauses. Dans la résolution SAT parallèle, le partage des clauses est un aspect critique qui nécessite une attention particulière (voir Section 2.3.3). Une implémentation sous-optimale du partage des clauses peut avoir un impact significatif sur l'efficacité du solveur, par exemple en raison d'une utilisation inappropriée des verrous ou de problèmes de synchronisation. Cette section présente l'organisation des mécanismes de partage dans `Painless`.

Le partage des clauses apprises est géré par un ou plusieurs threads appelés `Sharer`. Chaque `Sharer` est responsable d'un ensemble de producteurs et de consommateurs (qui sont des références à des `SolverInterface`). La stratégie de partage est définie en implémentant l'interface `SharingStrategy`. Le `Sharer` fonctionne dans une boucle simple : des phases de sommeil (d'une durée paramétrable) et des appels à la `SharingStrategy` pour effectuer le partage. Le développeur ne devrait pas avoir besoin de profondément modifier le `Sharer`. Pour définir une stratégie de partage, il suffit d'implémenter la méthode `doSharing()` de `SharingStrategy`, qui récupère les clauses des producteurs en appelant leurs méthodes `getLearnedClauses()` et les distribue aux consommateurs en appelant leurs méthodes `addLearnedClauses()`.

Une implémentation de `SharingStrategy` fournie de base dans `Painless` [94] est la stratégie de partage de `HordeSat`, appelée `HordesatSharing`. Cette classe maintient une base de données de clauses apprises pour chaque producteur, remplie par les appels à

`getLearnedClauses()`. À chaque tour de partage, la stratégie tente de récupérer jusqu'à 1500 littéraux (la somme de la taille des clauses partagées). Si la stratégie a du mal à récupérer suffisamment de littéraux (couramment moins de 75%), elle appelle la méthode `increaseClauseProduction()` du producteur concerné. Inversement, si la stratégie a du mal à vider la base de données de clauses apprises, `decreaseClauseProduction()` est appelée.

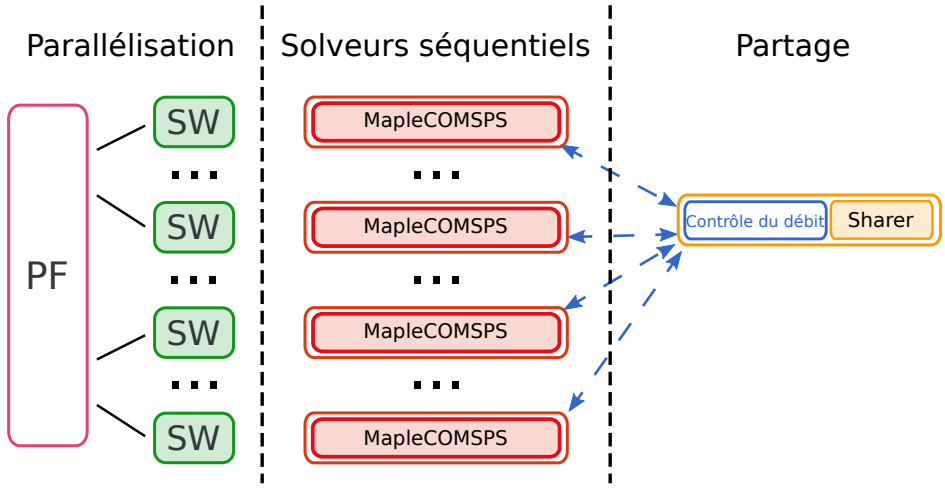


FIGURE 2.6 – configuration de `Painless` afin de créer le solveur parallèle portfolio `P-mcomsps`

Exemple d'instanciation : `P-mcomsps`. La Figure 2.6 présente une configuration de `Painless`, appelée `P-mcomsps`, soumise à la compétition de solveurs SAT parallèles 2020 [94]. `MapleCOMSPS` utilise une version de `SatElite` comme préprocesseur (Section 2.2.7). Le solveur peut utiliser le *VSIDS* comme le *LRB* pour décider de l'ordre de branchement ainsi que la *sauvegarde de la phase* pour le choix des valeurs (Section 2.2.5). Les clauses apprises sont stockées dans trois différentes bases selon leurs valeurs de LBD (Section 2.2.6). Les clauses avec une LBD inférieure ou égale à 3 ne sont jamais supprimées. Les clauses avec une LBD inférieure ou égale à 6 peuvent être rétrogradées à la dernière base si elles ne sont pas utilisées dans l'analyse de conflit pendant une longue période. Enfin, les autres clauses sont supprimées dès que le solveur décide de réduire son empreinte mémoire.

Sur la Figure 2.6 sont présents les trois composants de `Painless` définis ci-dessus :

— Parallélisation : La `WorkingStrategy` ici est un simple portfolio.

— Solveurs séquentiels : L'algorithme séquentiel utilisé est `MapleCOMSPS`. Ce solveur est diversifié en initialisant de manière aléatoire l'heuristique de choix de valeur, ici la sauvegarde de la phase (Section 2.2.5). L'heuristique de décision est fixée à *VSIDS* pour la moitié des travailleurs et à *LRB* pour l'autre moitié. Finalement, l'un des travailleurs exécute également l'élimination gaussienne (Section 2.2.7).

— Partage : Un thread Sharer, associé à une stratégie `HordesatSharing` accepte tous les solveurs du portfolio comme producteurs et consommateurs.

Chapitre

3

Politique de partage basée sur la structure des communautés

Contents

3.1 INTRODUCTION

Les performances des solveurs SAT parallèles modernes dépendent fortement de l'efficacité de leurs politiques de partage. Le problème abordé par ces politiques peut être résumé comme "le problème de l'identification de clauses apprises de haute qualité". Ces clauses, lorsqu'elles sont partagées entre les noeuds de travail d'un solveur parallèle, devraient mener à améliorer les performances. Le terme "clause de haute qualité" est souvent défini en termes de métriques que les concepteurs de solveur ont identifiées à travers des années d'études empiriques. Parmi les métriques les plus connues pour identifier ces clauses de haute qualité on peut noter : leurs tailles, leurs fréquences d'apparition dans la propagation unitaire ainsi que la LBD (voir Section 2.2.3). Le problème du LBD est sa localité, en effet la LBD est dépendante de l'état courant d'un solveur, une clause n'aura donc pas la même LBD d'un solveur à l'autre.

Dans ce chapitre, nous proposons une nouvelle métrique visant à identifier les clauses apprises de haute qualité et une politique concomitante de partage des clauses basée sur la combinaison de la LBD et la structure des communautés exposée par les formules SAT. Nous appelons un groupe de variables avec un lien fort entre elles et un lien faible avec le reste des variables une communauté. Une instance du problème SAT peut contenir des dizaines ou des milliers de communautés. Cette dernière est une propriété fréquente des instances encodant des problèmes du monde réel. Il est bien connu que les instances industrielles (en opposition à celles générées aléatoirement) ont des variables plus contraintes ensemble (liées par plus de clauses).

Le concept de structure des communautés a été proposé comme une explication possible pour l'extraordinaire performance des solveurs SAT sur des instances industrielles. De ce fait, cette structure est une candidate naturelle pour servir de base à une métrique permettant d'identifier les clauses de haute qualité. Pour être plus précis, notre métrique identifie les clauses qui ont une LBD et une valeur de communauté basse comme des clauses de qualité. La valeur de communauté d'une clause ω mesure le nombre de communautés différentes couvertes par les variables de cette clause. Nous effectuons une analyse empirique approfondie de notre métrique et notre politique de partage de clauses et nous montrons que notre méthode surpasse les techniques de l'état de l'art sur le benchmark provenant de la compétition de solveurs SAT parallèles.

Les contributions présentées dans ce chapitre sont les suivantes :

- En nous basant sur les statistiques recueillies durant l'exécution d'un solveur SAT sur le benchmark de la compétition SAT 2018, nous étudions la relation entre LBD et communauté et nous analysons l'efficacité de la LBD et de la valeur de communauté comme métrique de prédiction de l'utilité d'une clause nouvellement apprise.
- Sur la base de cette analyse, nous proposons de combiner les deux métriques pour en former une nouvelle et l'utiliser pour l'implémentation d'une nouvelle politique de partage de clauses dans un solveur SAT parallèle.
- Nous implémentons notre nouvelle stratégie de partage dans le solveur (`P-MCOMSPS` [59]) gagnant de la compétition 2018 et nous évaluons notre solveur sur le benchmark provenant de la compétition SAT 2016, 2017, 2018 et 2019.

Structure du chapitre. Nous introduisons les différentes manières de représenter une instance du problème SAT sous forme de graphe dans la Section 3.2, une notion utile pour définir une métrique basée sur la structure du problème. La Section 3.3 définie le concept de communauté et les algorithmes associés. Section 3.4 présente les analyses préliminaires sur l'utilisation de la LBD afin de fournir une intuition et une motivation à nos travaux. Section 3.5 explore l'idée de combiner la LBD et la valeur de communauté d'une clause pour détecter les clauses apprises utiles ou de haute qualité. Le solveur parallèle résultant et les résultats expérimentaux sont présentés dans Section 3.6. Finalement, Section 3.7 conclut ce chapitre.

3.2 Extraire la structure d'une formule SAT

Une instance du problème SAT peut être vue comme un hypergraphe où les arêtes sont une représentation des clauses liant les variables entre elles. Cette section présente différentes formes de graphe utiles et leurs avantages et inconvénients (et la figure associée 3.1). Puis nous présentons la représentation choisie pour calculer la valeur de communauté des clauses du problème.

3.2.1 *Hypergraphe*

La manière naturelle de représenter la structure d'une formule SAT est la construction d'un *hypergraphe*. Un hypergraphe est un graphe dont les arêtes peuvent lier plus de deux sommets. Pour une formule SAT φ donnée, son hypergraphe est un graphe dont les noeuds sont les variables de φ et les arêtes sont les clauses de φ. Chaque clause de taille k génère une arête dont les k points de terminaison sont les variables de k.

3.2.2 *Graphe d'incidence clause-variable*

Un hypergraphe peut être représenté comme un *graphe biparti*, *i.e.,* un graphe pour lequel l'ensemble des sommets peut être décomposé en deux parties disjointes de telle sorte qu'aucune arête ne relie deux sommets appartenant à la même partie. La transformation de l'hypergraphe d'une formule SAT en un graphe biparti requiert la création d'un *graphe d'incidence clause-variable (GICV)* [2] de cette formule. Soit la formule φ, le GICV est le graphe biparti dont les sommets représentent l'ensemble des variables de φ et l'ensemble des clauses de φ. Il existe un lien entre une clause et une variable si la variable est contenue dans la clause.

Afin de revenir à un simple graphe, un graphe biparti peut être projeté sur un de ses deux ensembles de sommets. Dans notre cas, nous pouvons projeter ce graphe sur les sommets représentant les variables de la clause.

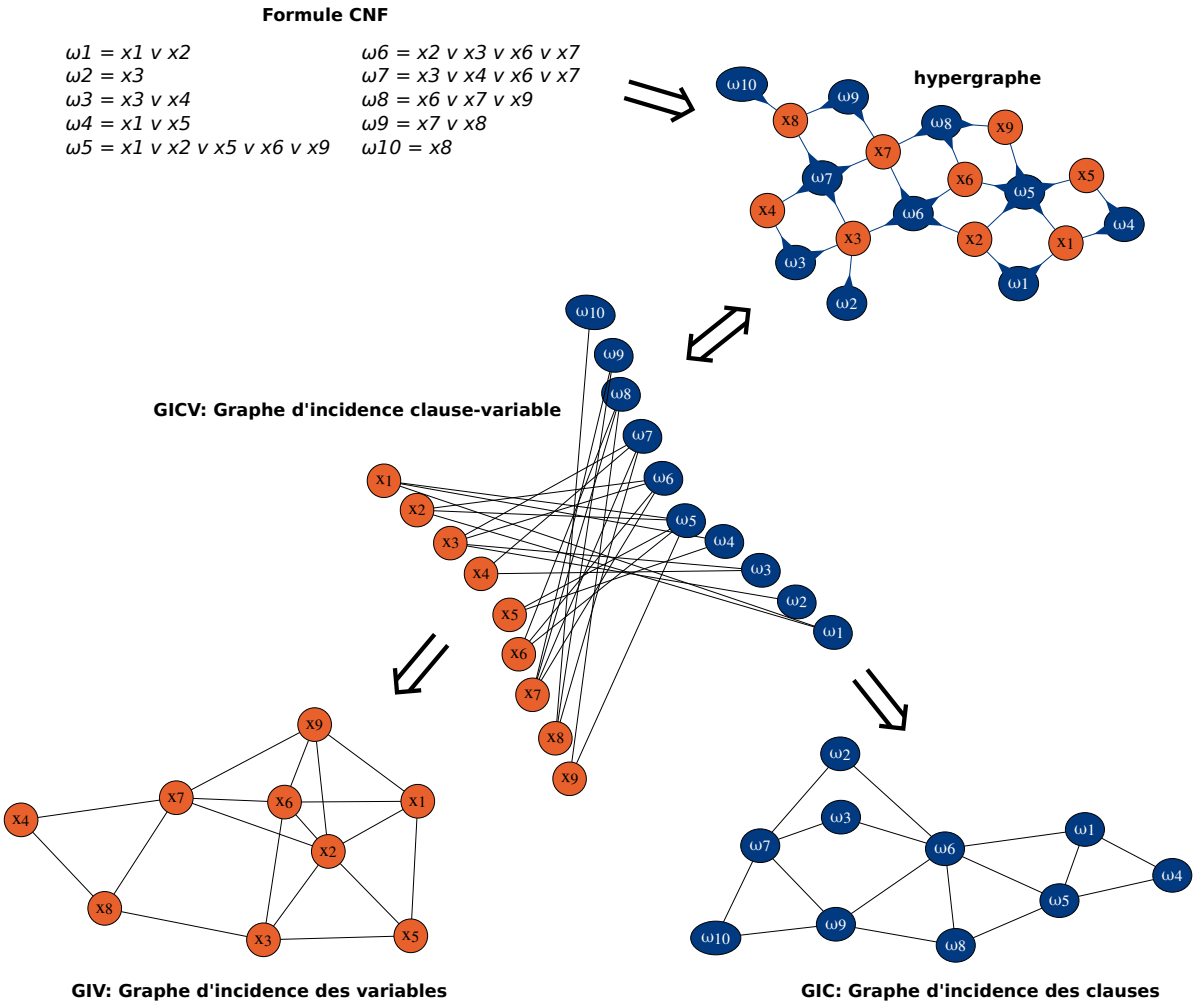


FIGURE 3.1 – Transformation d'une formule CNF vers différentes représentations de graphes

3.2.3 *Graphe d'incidence des variables*

Lorsque nous projetons sur les variables, nous générons le *graphe d'incidence des variables (GIV)* [2]. Soit la formule φ, le GIV de φ est le graphe dont les sommets représentent les variables de φ et où il existe une arête entre deux variables si elles apparaissent toutes deux dans une clause $\omega \in \varphi$. Cette forme ne conserve pas d'informations sur la taille des clauses et le nombre de fois qu'une paire de variables se trouvent dans la même clause. Une solution permettant d'intégrer ces informations à la structure du graphe est de définir une fonction de pondération qui indique pour chaque arête la force du lien entre deux variables. L'idée est de renforcer le poids d'une arête si deux variables apparaissent souvent ensemble dans des clauses de petite taille. En effet, le lien entre une paire de variables n'est pas le même si elles apparaissent dans une clause de taille 2 ou une clause de taille 20. D'après la définition du GIV, une clause ω donne lieu à la formation de $\binom{|\omega|}{2}$ arêtes. Afin d'accorder la même importance à chaque clause, nous fixons leurs poids à $\frac{1}{\binom{|\omega|}{2}}$. Ainsi, nous obtenons la formule de pondération

suivante :

$$w(x_1, x_2) = \sum_{\substack{\omega \in \varphi \\ x_1, x_2 \in \omega}} \frac{1}{\binom{|\omega|}{2}} \tag{3.1}$$

Notons qu'il est également possible de choisir une représentation basée sur les littéraux au lieu des variables, nommée graphe d'incidence des littéraux (GIL). Cette représentation suit la même logique que celle décrite ici et aura deux fois plus de sommets que le GIV, dans le pire des cas, mais renforce la précision de la relation entre les sommets.

3.2.4 *Graphe d'incidence des clauses*

Alternativement, projeter sur les clauses génère un *graphe d'incidence sur les clauses (GIC)*. Soit la formule φ, le GIC de φ est le graphe dont les sommets représentent des clauses de φ et il existe une arête entre deux clauses si elles partagent une variable x dans φ. Ici, l'information perdue est la fréquence d'apparition d'une variable. Comme pour le GIV, nous pouvons également définir une fonction de pondération pour intégrer cette information à la structure du graphe. Nous voulons renforcer le poids d'une arête si les clauses sont liées par des variables qui n'apparaissent pas souvent dans le reste de la formule. Nous définissons $\mathtt{cls}(x)$ comme une fonction retournant le nombre de clauses dans lesquelles la variable x apparaît. Une variable x apparaissant dans $\mathtt{cls}(x)$ clauses génère $\binom{|\mathtt{cls}(x)|}{2}$ arêtes. Soit x une variable, ω_1 et ω_2 une paire de clauses telles que $x \in \omega_1 \cap \omega_2$. Le poids de x pour l'arête reliant ω_1 et ω_2 est donné par $\frac{1}{\binom{|\mathtt{cls}(x)|}{2}}$. Ainsi le poids d'une variable donnée sur une arête dépend de sa fréquence d'apparition dans le reste de la formule. La formule de pondération est donc :

$$w(\omega_1, \omega_2) = \sum_{\substack{x \in \omega_1 \cap \omega_2 \\ \omega_1, \omega_2 \in \varphi}} \frac{1}{\binom{|\mathtt{cls}(x)|}{2}} \tag{3.2}$$

3.2.5 *Discussion des différentes représentations*

Étant donné que les formules traitées par les solveurs SAT peuvent comporter des millions de variables et des dizaines de millions de clauses, leurs représentations sous forme de graphe peuvent être très volumineuses et doivent être choisies avec soin. En considérant le nombre de sommets, les graphes suivants sont présentés dans l'ordre croissant de leur probabilité à avoir le plus de sommets : GIV, GIL, GIC, GICV. Les graphes plus petits (*i.e.,* GIV et GIC) perdent plus d'informations lors de la projection.

Le choix de représentation doit être considéré en fonction du niveau de précision souhaité. Dans le cadre de la résolution SAT, cette information est souvent utilisée de manière heuristique, il n'est donc pas grave d'en tirer des informations moins précises. Pour le reste du

chapitre, nous nous concentrons sur le GIV qui est le plus petit en termes de nombre de sommets, mais qui permet de conserver suffisamment d'informations notamment grâce au poids sur les arêtes.

3.3 COMMUNAUTÉ

Il est bien admis que les formules SAT provenant de problèmes réels (comprendre non aléatoire) présentent une structure particulière, ce qui explique le succès de certaines heuristiques telles que le VSIDS [74] ou la sauvegarde de la phase [81], par exemple. Une façon de mettre en évidence une telle structure est de représenter la formule sous forme de graphe et d'analyser sa structure. En particulier, c'est la *structure de communauté* [2] exposée par certains graphes qui nous intéresse dans ce chapitre. Dans un graphe G, des communautés sont des sous-graphes de G très denses et faiblement reliés au reste du graphe. Cette section définit comment cette structure est communément extraite d'un graphe.

Un *graphe pondéré non dirigé* est une paire $G = (N, w)$, où N est l'ensemble des sommets de G, et $w : N \times N \to \mathbb{R}^+$ est la fonction de poids commutative associée. La structure des communautés d'un tel graphe est généralement extraite à l'aide de la métrique de modularité [78].

3.3.1 *Modularité*

La fonction de modularité $\mathcal{Q}(G, P)$ (Équation (3.3)), prend un graphe G et une partition $P = \{P_1, \ldots, P_n\}$ de sommets de G. Elle évalue la densité de la connexion des sommets au sein d'un sous-graphe par rapport à la densité de l'ensemble du graphe. Le but est de trouver une partition du graphe qui maximise cette valeur.

$$\mathcal{Q}(G, P) = \sum_{P_i \in P} \frac{\sum_{x,y \in P_i} w(x, y)}{\sum_{x,y \in N} w(x, y)} \tag{3.3}$$

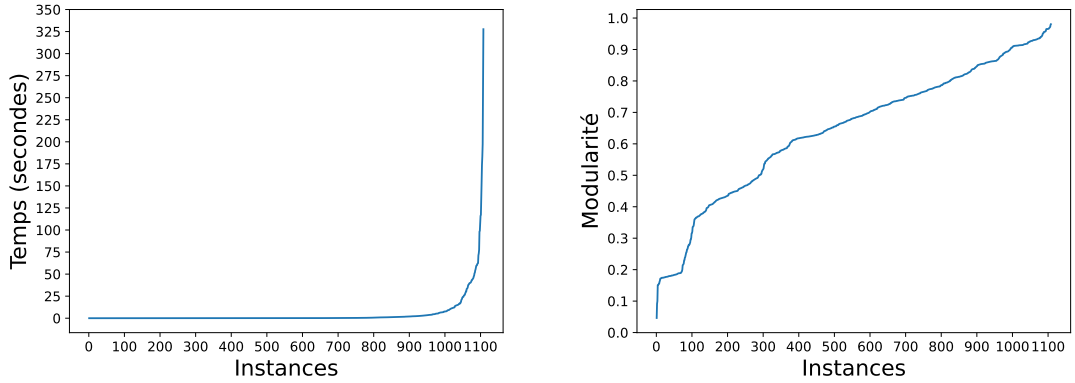
Le problème de cette définition de la modularité est qu'une partition qui contient tous les sommets du graphe maximise effectivement cette valeur. De ce fait, nous soustrayons à la première formule la valeur obtenue pour une partition dans un graphe aléatoire avec le même nombre de sommets et le même degré pour chaque sommet (Équation (3.4)).

$$\mathcal{Q}(G, P) = \sum_{P_i \in P} \frac{\sum_{x,y \in P_i} w(x, y)}{\sum_{x,y \in N} w(x, y)} - \left(\frac{\sum_{x \in P_i} \deg(x)}{\sum_{x \in N} \deg(x)} \right)^2 \tag{3.4}$$

On appelle alors **modularité d'un graphe** G la modularité maximale pour toutes partitions possibles de ses sommets P : $Q(G) = \max\{\mathcal{Q}(G, P) \mid P\}$. Elle est donc comprise entre $[0, 1]$.

3.3.2 *Algorithme de Louvain*

Calculer la modularité d'un graphe est un problème NP-difficile [24]. Cependant, il existe des algorithmes gloutons efficaces, renvoyant une borne inférieure approximative pour la modularité d'un graphe, comme la *Méthode de Louvain* [21]. Dans ce chapitre, nous utilisons la méthode de Louvain (avec une précision $\epsilon = 10^{-7}$) pour extraire les communautés. La représentation choisie sur laquelle nous appliquons cette méthode et le GIV. Ces graphes sont créés à partir de formules déjà simplifiées par le préprocesseur `SatElite` [38] (voir Section 2.2.7), utilisé par `P-mcomsps` au démarrage.



(a) Durée d'exécution de Louvain par instances **(b)** Modularité des instances

FIGURE 3.2 – Louvain sur la compétition SAT : temps de calcul et modularité

La Figure 3.2a montre la durée en secondes nécessaire à la méthode Louvain pour extraire les communautés des formules de notre benchmark. Ce benchmark comprend les 115O instances provenant de la compétition de solveurs SAT 2017, 2018 et 2019. Cette expérience a été réalisée avec un temps limite de 5000 secondes. Nous observons que Louvain est peu coûteux en calcul pour la grande majorité des instances. En effet, deux tiers des instances nécessitent moins d'une seconde et la plupart du dernier tiers une dizaine de secondes. Cependant, nous notons qu'il y a des instances pour lesquelles Louvain prend quelques minutes, et en particulier, ne retourne pas de résultat pour 16 d'entre elles et explose en mémoire pour une (sur une machine avec 64Go de mémoire).

La Figure 3.2b montre la modularité obtenue après la fin de Louvain. On peut voir que deux tiers de notre benchmark expose une modularité intéressante avec une valeur supérieure à 0.5. Cependant, ce dernier tiers du benchmark sans structure de communauté particulière, en plus des quelques cas où la structure est difficile à extraire, suggère que l'exploitation des communautés dans un contexte séquentiel est risquée. Au contraire, dans un contexte parallèle, il peut être intéressant de profiter des multiples cœurs disponibles pour calculer les communautés en parallèle à la résolution, afin de ne pas entraver les performances.

3.3.3 *La valeur de communauté d'une clause*

Nous appelons COM le nombre de communautés sur lesquelles une clause s'étend : nous travaillons sur le GIV de la formule, nous considérons donc des communautés de variables. Chaque variable appartient à une communauté unique (déterminée par l'algorithme de Louvain). Pour calculer COM, nous considérons les variables correspondant aux littéraux de la clause et nous comptons le nombre de communautés distinctes représentées par ces variables.

3.3.4 *Exploitation des communautés*

La structure de communauté notable des formules SAT industrielles a été identifiée dans [2]. Newsham et al. pensent qu'une telle structure est l'une des principales raisons des performances notables des solveurs SAT sur les problèmes industriels [79]. Ainsi, plusieurs travaux exploitent les communautés pour améliorer les performances des solveurs [71, 90]. Nous notons particulièrement l'exploitation de cette structure par Ansótegui et al. [3] qui ont développé un préprocesseur (Section 2.2.7) appelé `modprep`. Ce préprocesseur injecte dans la formule des clauses avec une faible valeur COM. Pour ce faire, `modprep` divise la formule en communautés de clauses. Chaque paire de sous-formules dont l'intersection des variables n'est pas vide est alors résolue par un solveur CDCL classique. Les clauses apprises par ce solveur sont ensuite ajoutées pour étendre le problème initial. Cette technique a montré de bonnes performances, même en tenant compte du temps de calcul des communautés et de la résolution des sous-formules. Elle reste cependant coûteuse et se limite aux clauses apprises lors de la phase de preprocessing. Nous voulons exploiter le parallélisme pour développer cette idée plus en profondeur.

3.4 MESURE DE L'EFFICACITÉ DU LBD ET SES LIMITES

Notre objectif principal est d'améliorer les performances globales de la résolution SAT en parallèle en augmentant la qualité du partage des clauses apprises. Pour se faire, nos travaux se concentrent sur le raffinement du critère de sélection des clauses à partager. Dans la plupart des solveurs SAT parallèles concurrents, le partage est limité à des formes particulières de clauses apprises (clauses *unitaires*, clauses avec une LBD $\leq$ i [15, 59]). Nous proposons dans un premier temps de centrer notre étude sur la LBD et d'évaluer l'impact de partager des clauses selon cette heuristique.

Une manière d'y parvenir est d'étudier l'impact des clauses partagées entre les moteurs SAT sous-jacents (solveurs SAT séquentiels) pour une stratégie de partage donnée. Puisque déterminer, a priori, l'utilité de l'information est déjà une tâche difficile dans un contexte séquentiel, on peut facilement imaginer la difficulté d'établir cette estimation dans le cadre parallèle. Nous proposons donc une méthodologie consistant dans un premier temps à sélectionner une valeur pour notre métrique en analysant son évolution selon un marqueur d'utilité dans un contexte séquentiel. Puis, évaluer les performances de la stratégie de partage dérivée dans un contexte

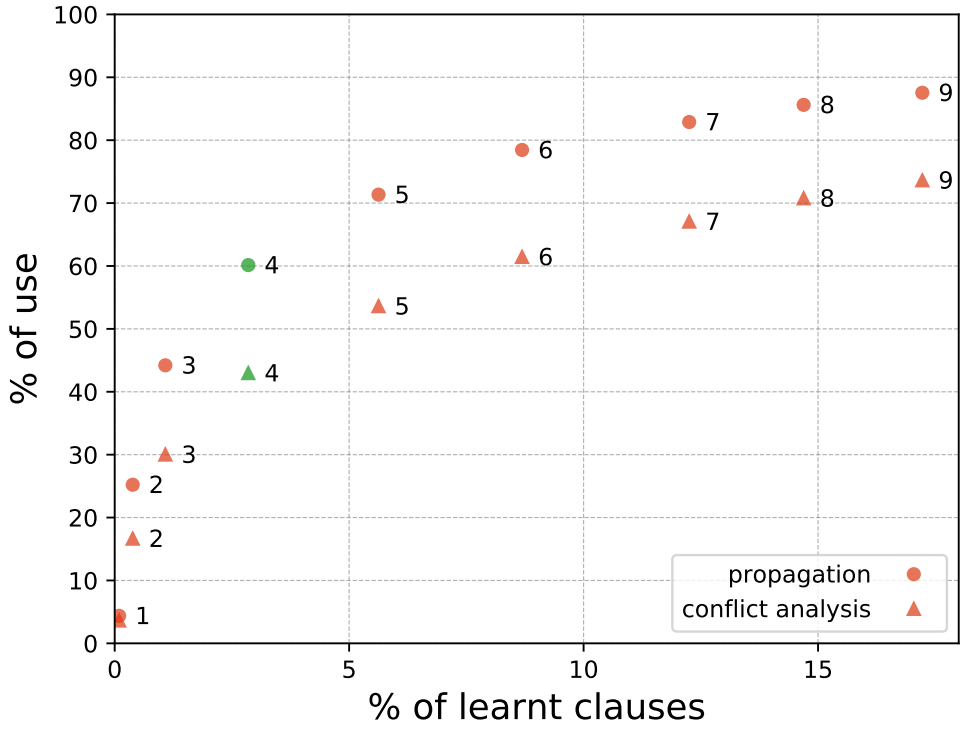


FIGURE 3.3 – Utilisation des clauses apprises selon leurs LBD

parallèle. Fixer une valeur pour la métrique grâce à une évaluation séquentielle permet d'éviter le non-déterministe introduit par le parallélisme, mais reste logique, car les solveurs parallèles lancent plusieurs solveurs séquentiels.

Nous désignons comme marqueur de l'utilité d'une clause son utilisation dans deux des composants principaux d'un solveur séquentiel, à savoir les procédures de *propagation unitaire* et d'*analyse de conflit* (voir Sections 2.2.1 et 2.2.3).

3.4.1 *Résolution séquentielle SAT, clauses apprises et LBD*

Pour mener cette étude, nous exécutons `MapleCOMSPS` [62] sur les instances du concours SAT 2018 [1] avec une limite de temps de 5000 secondes. Figure 3.3 illustre nos observations. L'axe des abscisses montre le pourcentage cumulatif des clauses apprises (en considérant les clauses apprises de toutes les exécutions). L'axe des ordonnées correspond au pourcentage d'apparition dans un événement donné, également en cumulatif. Par conséquent, la courbe avec des points représente, pour différentes LBD (de 1 à 9), l'utilisation de ces clauses dans la propagation unitaire. Dans la même figure, la courbe tracée avec des triangles met en évidence les mêmes informations, mais pour l'analyse de conflit.

Dans les deux courbes, l'observation clé concerne l'inflexion située autour de LBD=4. L'impact des clauses avec LBD ≤ 4 peut être considéré comme très positif : 60% de l'utilisation

1. http://sat2018.forsyte.tuwien.ac.at

dans la propagation unitaire, et 40% dans l'analyse de conflit, tout en ne représentant que 3% du nombre total de clauses apprises. De plus, les clauses avec un LBD > 4 n'apportent pas une valeur ajoutée significative lorsque l'on considère leurs quantités.

Sur la base de ces résultats, 4 semble être une bonne valeur LBD pour limiter le taux de clauses partagées tout en maximisant la fréquence de ces deux événements dans un contexte parallèle.

3.4.2 *Une première stratégie de partage parallèle*

Pour évaluer notre observation précédente, nous avons développé une stratégie mettant en œuvre un *partage de clauses basé sur la LBD* : les clauses apprises avec une LBD en dessous d'un seuil prédéterminé sont partagées. Nous avons ensuite opéré cette stratégie avec LBD=4, mais aussi avec les valeurs environnantes (LBD=3 et LBD=5). Notre stratégie a été intégrée dans `Painless` [57] en utilisant `MapleCOMSPS` comme moteur de solveur séquentiel et une stratégie de parallélisation de type Portfolio. Les solveurs que nous comparons sont :

— `P-MCOMSPS`, solveur de type Portfolio gagnant de la compétition de solveur SAT parallèle en 2018. Il est le point de référence pour cette évaluation. Il implémente une stratégie de partage dynamique qui peut augmenter le seuil de LBD pour maintenir un débit de clauses partagées prédéterminées [59] (Section 2.3.4).

— `P-MCOMSPS-L`$\langle n \rangle$, notre nouveau solveur, avec LBD $\leq n$.

Tous les solveurs ont été exécutés sur une machine possédant un Intel Xeon CPU E5645 de 12 cœurs et une mémoire vive de 64 Go. Ils ont été lancés avec 12 threads, une limite de mémoire de 61 Go et une limite de temps de 5000 secondes (comme pour les compétitions SAT).

Solveurs	2017		2018		2019	
	PAR-2	# instances résolues	PAR-2	# instances résolues	PAR-2	# instances résolues
`P-MCOMSPS`	**355h42**	**237**	430h13	258	**380h03**	**273**
`P-MCOMSPS-L3`	361h27	234	420h53	263	393h04	269
`P-MCOMSPS-L4`	356h44	**237**	**411h14**	**265**	391h38	269
`P-MCOMSPS-L5`	369h27	229	415h52	264	389h27	269

TABLE 3.4 – Comparaison des performances pour différents seuils de LBD (3, 4, and 5). `P-MCOMSPS` est utilisé comme référence.

La Table 3.4 présente nos mesures sur les instances des concours SAT 2017, SAT 2018 et SAT 2019. PAR-k (Penalized average runtime) est le temps d'exécution moyen, ajoutant une pénalité de k fois la limite de temps d'exécution pour chaque exécution ne finissant pas dans les délais imposés. La valeur utilisée pour k dans le concours annuel SAT est de 2. Les cellules grisées indiquent quel solveur a les meilleurs résultats pour une suite d'instances donnée. Cela montre que la stratégie basée sur un LBD ≤ 4 n'est pas aussi efficace que nous pourrions

nous y attendre. En particulier, nous notons une grande instabilité en fonction des ensembles d'instances à traiter.

Nous pensons que ces résultats sont dus au fait que la LBD est trop liée à l'état local d'un solveur. L'intuition que nous étudions dans ce chapitre est que la métrique LBD doit être renforcée avec des informations plus globales.

3.5 Conception d'une métrique hybride

Comme indiqué précédemment, nous avons besoin d'une métrique indépendante de l'état local d'un solveur particulier. Des informations structurelles sur la formule à résoudre peuvent être utiles. Par exemple, dans un solveur de type Portfolio, les informations structurelles peuvent être partagées entre les solveurs travaillant sur la même formule, sans avoir besoin de considérer qu'ils sont à différents endroits de l'espace de recherche.

Dans ce chapitre, nous nous concentrons sur la structure de communauté exposée par les instances provenant de problèmes non générés aléatoirement. Il a été prouvé que les métriques (COM, définies dans la Section 3.3.3) dérivées de cette structure sont liées à la LBD dans [79]. En outre, elle a été utilisée pour améliorer les performances de la résolution SAT séquentielle via une approche de prétraitement dans [3].

Cette section montre que les communautés sont de bonnes candidates pour fournir l'information globale souhaitée. Pour ce faire, nous utilisons le même protocole que celui utilisé dans la section précédente : *(i)* étudier les données sur la résolution SAT séquentielle pour présenter de bons candidats pour les valeurs des paramètres, puis *(ii)* vérifier son efficacité dans un contexte parallèle. Nous avons complété les logs extraits des expériences séquentielles présentées dans Section 3.4 avec des informations sur les communautés et avons étudié l'ensemble comme suit.

3.5.1 *LBD versus Communautés*

Tout d'abord, nous avons étudié la relation entre les valeurs de LBD et COM grâce à deux heatmap (Figure 3.5). Figure 3.5a montre, pour chaque valeur de LBD, la distribution des valeurs COM. Figure 3.5b montre, pour chaque valeur COM, la distribution des valeurs LBD. Par exemple, en analysant la Figure 3.5a, nous observons que $\approx$ 65% des clauses avec LBD=1 s'étendent sur une communauté (COM =1), $\approx$ 20% d'entre elles s'étendent sur deux communautés (COM =2), etc. À partir de ces chiffres, nous pouvons conclure deux affirmations importantes :

— Figure 3.5a, les zones chaudes sur la diagonale pour les valeurs LBD faibles indiquent une sorte de corrélation entre les valeurs LBD et COM (un résultat qui a déjà été présenté dans [79]). En regardant de plus près, on observe qu'une partie importante des clauses ne suit pas cette corrélation : les zones chaudes restent en dessous de 65%,

et se situent principalement en dessous de 25%. Par conséquent, les métriques COM semblent être un bon candidat pour affiner les clauses déjà sélectionnées à l'aide de LBD ;

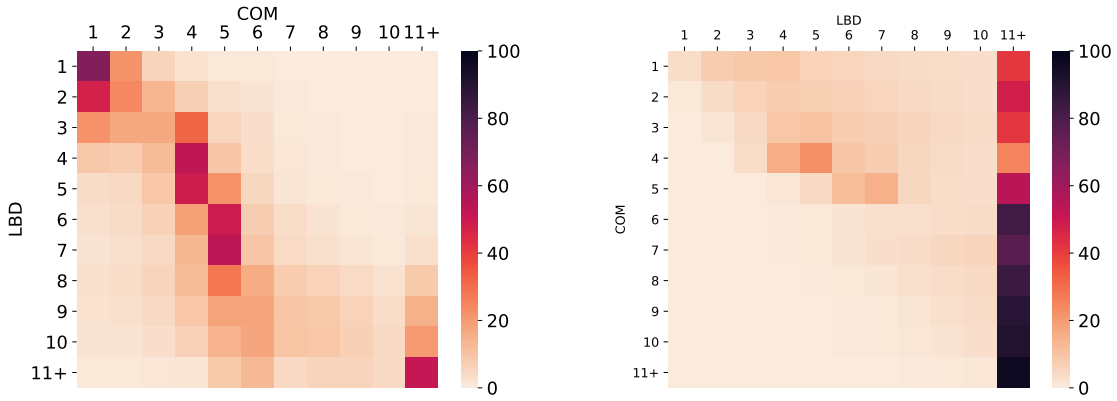
— Figure 3.5b, les valeurs COM sont réparties presque uniformément sur toutes les valeurs LBD. Nous concluons que l'utilisation de COM comme seule mesure de sélection est une mauvaise idée (nous avons évalué cela avec plusieurs expériences avec des solveurs parallèles qui ne sont pas présentés dans ce chapitre).

Comme indiqué précédemment, COM est un critère complémentaire au LBD. Il faut donc découvrir les bons couples de valeurs qui maximisent l'utilisation des clauses partagées, tout en conservant un débit de partage raisonnable.

3.5.2 *Composition du LBD et des communautés*

Tout d'abord, nous notons à partir de la Figure 3.3 que les clauses avec LBD ≤ 3 représentent 1,1% du total des clauses pour un pourcentage d'utilisation allant jusqu'à 44,2%. Il n'y a donc aucun avantage à filtrer ces clauses. D'ailleurs, ce premier ensemble de clauses n'est pas suffisant puisque la Table 3.4 montre que le solveur parallèle avec un seuil LBD=3 ne gagne jamais. Par conséquent, nous recherchons l'ensemble des clauses qui peuvent être ajoutées à LBD ≤ 3 pour améliorer les performances. Sur la base des observations faites dans la Section 3.4, nous pensons que le meilleur candidat est un sous-ensemble de LBD = 4 ou LBD = 5. Pour identifier le ou les bons couples, nous utilisons notre protocole d'expérimentation et analysons les données recueillies pour LBD et COM. Par conséquent, nous recherchons un sous-ensemble de clauses dans LBD > 3 qui profite à la propagation unitaire et l'analyse de conflit.

Pour capturer l'utilité des clauses apprises, nous définissons le **ratio d'utilisation** comme suit : le rapport entre le pourcentage d'utilisation (propagation unitaire ou analyse de conflit)



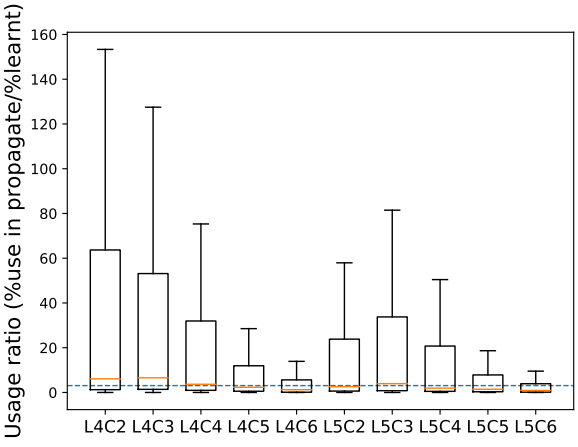
(a) Distribution de COM pour différentes LBD **(b)** distribution de LBD pour différentes COM

FIGURE 3.5 – Heatmap montrant la distribution entre COM et LBD

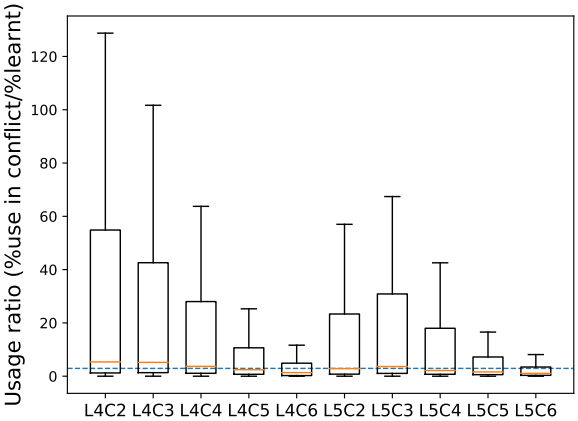
et le pourcentage de clauses apprises (parmi celles d'une formule donnée). Par exemple, dans une formule donnée, une clause avec un *ratio d'utilisation* de 10 est utilisée 10 fois plus que l'utilisation moyenne de toutes les clauses apprises. Par extension, le *ratio d'utilisation d'un ensemble de clauses* est la moyenne du ratio d'utilisation de toutes ses clauses.

Les données résultantes sont affichées sous forme de diagrammes de quartiles dans la Figure 3.6. Ces diagrammes intègrent la distribution du ratio d'utilisation pour toutes les formules du concours SAT 2018. Un diagramme noté L⟨x⟩C⟨y⟩ correspond à l'ensemble des clauses avec LBD=x et COM=y.

Nous observons que pour une valeur fixe de LBD, le ratio d'utilisation varie fortement en fonction des différentes valeurs de COM. Deuxièmement, nous discernons que les clauses avec LBD = 4 ont globalement un meilleur ratio d'utilisation que celles avec LBD = 5. Finalement, nous pouvons extraire les configurations les plus prometteuses, en remarquant que les configurations L4C2 et L4C3 ont le ratio d'utilisation le plus impressionnant : dans 25% (le 3^{th} quartile) des instances traitées, les clauses dans ces configurations ont un ratio d'utilisation supérieur à 50 dans la propagation unitaire (Figure 3.6a), 40 dans l'analyse de conflit (Figure 3.6b) et s'étendent à des valeurs très élevées (jusqu'à 150 en propagation et 130 en analyse de conflit). De plus, la médiane du taux d'utilisation de la propagation pour L4C2 et L4C3 (6,0 et 6,5, respectivement) est deux fois plus grande que la moyenne de l'ensemble des LBD = 4 et LBD = 5 cases (égale à 2,9 et notée par une ligne pointillée dans les deux figures).



(a) Ratio d'utilisation dans la propagation unitaire



(b) Ratio d'utilisation dans l'analyse de conflit

FIGURE 3.6 – Efficacité de la combinaison de la LBD et de COM. La ligne bleue en pointillé qui traverse les différents diagrammes correspond à la moyenne et la ligne orange à l'intérieur d'un diagramme à la médiane.

3.5.3 *Filtrage basé sur les communautés*

De cette étude, nous pouvons conclure que nous pouvons utiliser la structure de communauté comme filtre pour les clauses qui ont déjà été sélectionnées à l'aide d'un seuil sur la LBD.

Nous proposons la stratégie suivante :

— partager toutes les clauses avec un LBD ≤ 3 (sans aucune limite sur COM). En effet, cet ensemble de clauses est petit tout en étant très utile.

— partager toutes les clauses avec une LBD ≤ 4 et COM ≤ 3. Ces clauses permettent d'obtenir un ratio d'utilisation plus élevé tout en maintenant le partage à un débit raisonnable.

Pour vérifier cette affirmation, et valider l'efficacité du seuil choisi (COM ≤ 3), nous étendons l'étude présentée dans la Section 3.4.1. Ainsi, les Figure 3.7 et Figure 3.8 reprennent les mêmes points que la Figure 3.3, cette fois en séparant les résultats sur la propagation unitaire et sur l'analyse de conflit (pour augmenter la lisibilité). Les nouvelles figures ajoutent, sous la forme de triangles, le ratio d'utilisation pour différentes valeurs de filtre.

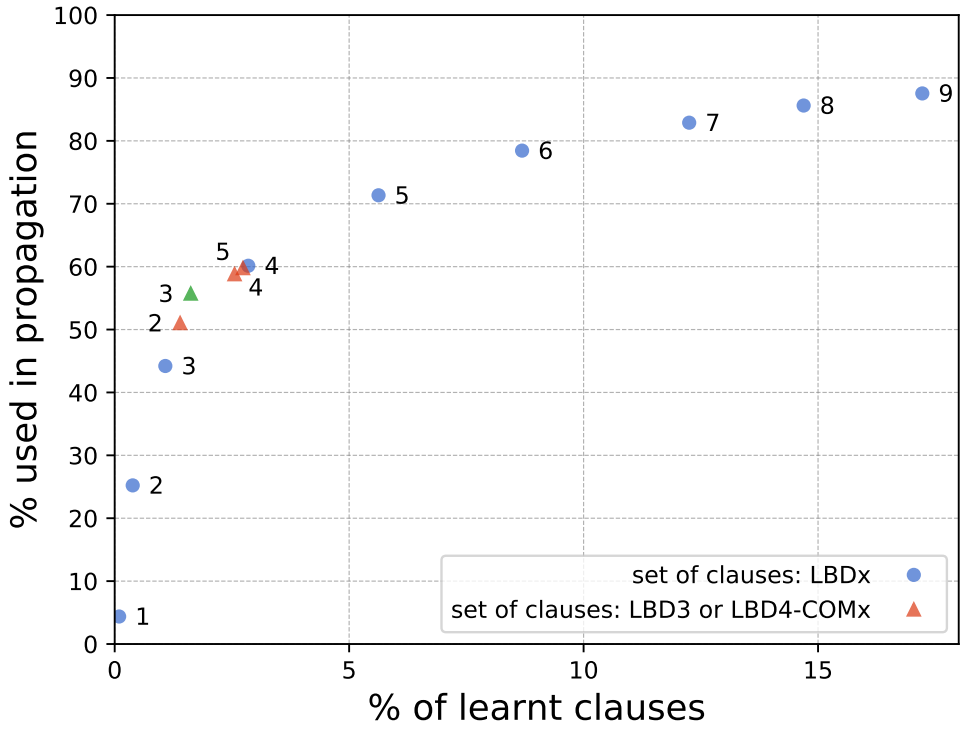


FIGURE 3.7 – Utilisation des clauses apprises dans la propagation unitaire en considérant la LBD

Comme prévu, le point COM = 3 (triangle vert) est à l'inflexion des courbes. Par conséquent, la sélection de cet ensemble de clauses nous permet d'atteindre un meilleur ratio d'utilisation, proche du LBD non filtré ≤ 4, tout en préservant un nombre comparable de clauses avec LBD ≤ 3. Cela nous convainc que ces mesures devraient conduire à une augmentation des

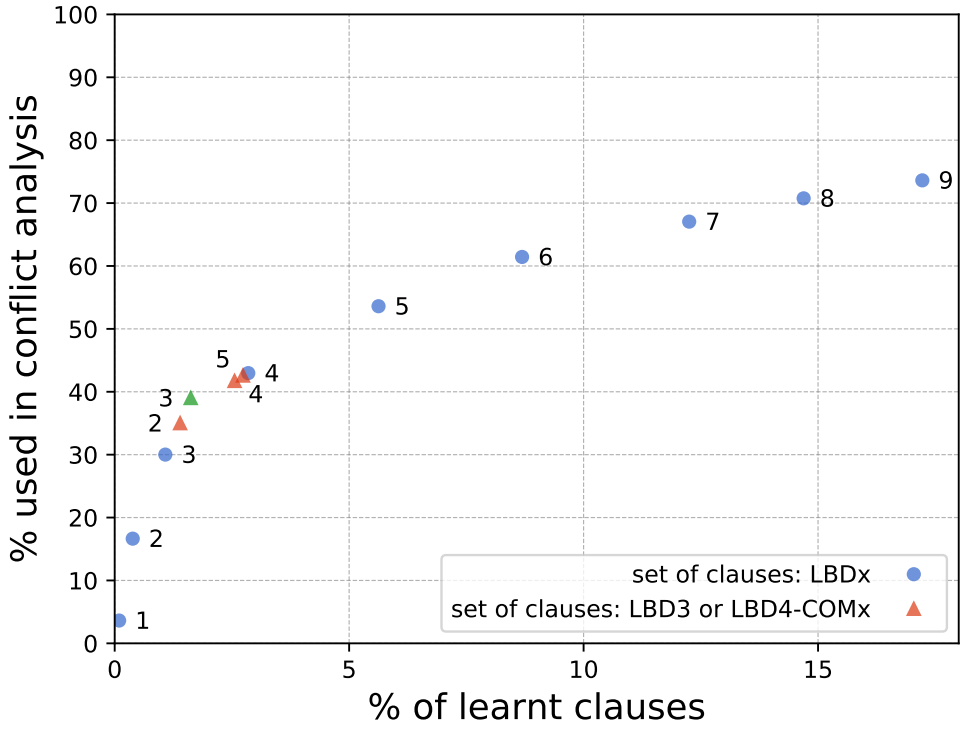


FIGURE 3.8 – Utilisation des clauses apprises dans l'analyse de conflit en considérant la LBD

performances dans la résolution SAT parallèle. La section suivante vérifie ces mesures dans la pratique.

3.6 ÉVALUATION DE LA NOUVELLE STRATÉGIE DE PARTAGE

Cette section décrit les solveurs SAT parallèles que nous avons conçus pour évaluer notre stratégie, ainsi que le protocole expérimental associé. Elle présente et discute ensuite nos résultats expérimentaux.

3.6.1 *Solveurs et protocole d'évaluation*

Comme dans la Section 3.4.2, nous utilisons le solveur `P-MCOMSPS`, sur la même plateforme matérielle, comme référence pour valider notre stratégie. La seule différence entre le solveur original et le nouveau réside dans leurs stratégies de partage. Celles-ci sont :

— `P-MCOMSPS` : la stratégie originale, basée sur un seuil de LBD dynamique.

— `P-MCOMSPS-L4C3` : seules les clauses apprises avec une LBD ≤ 3 ou LBD $= 4$ et COM ≤ 3 sont partagées.

Dans P-MCOMSPS-L4C3, une instance de solveur séquentiel est dédiée au calcul de la structure de communauté (à l'aide de l'algorithme de Louvain). En parallèle, les autres solveurs exécutent l'algorithme CDCL pour résoudre la formule et partagent les clauses avec un LBD ≤ 3. Comme indiqué dans la Section 3.5.2, le partage de toutes ces clauses ne doit pas altérer les performances. Une fois que les communautés ont été calculées, le solveur peut commencer l'algorithme CDCL (comme les autres), et la stratégie de partage initial est complétée par des clauses caractérisées par un LBD $= 4$ et $COM \leq 3$. Comme démontré dans la Section 3.3.2, il ne faut pas plus d'une minute pour terminer Louvain pour presque toutes les instances de tous les benchmarks. Par conséquent, le filtre augmenté est activé tôt dans la résolution d'une formule.

Pour l'évaluation, nous avons utilisé la suite d'instances des compétitions SAT 2016 [2], 2017 [3], 2018 [4] et 2019 [5]. Tous les solveurs ont été lancés sur les mêmes machines et avec la même configuration que dans la Section 3.4.2. Les résultats que nous avons observés sont discutés dans la section suivante.

3.6.2 *Évaluation du nouveau solveur parallèle*

Figure 3.9 présente les résultats obtenus. Lorsque l'on considère le nombre d'instances résolues, nous observons clairement que la nouvelle stratégie de partage surpasse celle utilisée dans P-MCOMSPS, pour toutes les suites d'instances de la compétition. Nous ajoutons que P-MCOMSPS-L4C3 résout 29 nouvelles instances par rapport à P-MCOMSPS et ne parvient pas à résoudre 16 instances que ce dernier résout.

En ce qui concerne le PAR-2, les résultats sont moins marqués. Nous trouvons une explication dans le comportement de la stratégie de partage de P-MCOMSPS : P-MCOMSPS démarre la résolution avec un seuil LBD bas et incrémente ce seuil s'il estime que le débit des clauses partagées n'est pas suffisant. Cette stratégie incrémentielle peut aider le solveur à apprendre des informations pertinentes menant à la résolution de certains cas particuliers. Au contraire, notre stratégie de partage restrictive peut manquer ces informations pertinentes pour ces cas particuliers.

Enfin, notre stratégie étant basée sur l'étude réalisée sur un solveur séquentiel, nous souhaitons vérifier l'évolution du ratio d'utilisation dans le contexte parallèle. Nous effectuons une nouvelle évaluation : en utilisant le même protocole que celui développé dans le cadre séquentiel (Section 3.5.2), nous calculons le ratio d'utilisation des clauses dans la propagation unitaire et l'analyse de conflit de notre solveur parallèle P-MCOMSPS-L4C3. Le nombre résultant est la somme de tous les moteurs CDCL séquentiels sous-jacents. Ces calculs sont effectués depuis 100 instances tirées aléatoirement des instances de la compétition SAT 2018.

2. https://baldur.iti.kit.edu/sat-competition-2016/downloads/app16.zip
3. https://baldur.iti.kit.edu/sat-competition-2017/benchmarks/Main.zip
4. https://satcompetition.github.io/2018/downloads.html
5. http://satcompetition.org/sr2019benchmarks.zip

	Solveurs	PAR-2	# instances résolues
2019	P-MCOMSPS-L4C3	386h14	**274**
	P-MCOMSPS	**380h03**	273
2018	P-MCOMSPS-L4C3	**408h01**	**268**
	P-MCOMSPS	430h13	258
2017	P-MCOMSPS-L4C3	**352h31**	**238**
	P-MCOMSPS	355h42	237
2016	P-MCOMSPS-L4C3	355h08	**183**
	P-MCOMSPS	**354h39**	182
All together	P-MCOMSPS-L4C3	**1 501h54**	**963**
	P-MCOMSPS	1 520h37	950

TABLE 3.9 – Évaluation des performances de P-MCOMSPS-L4C3

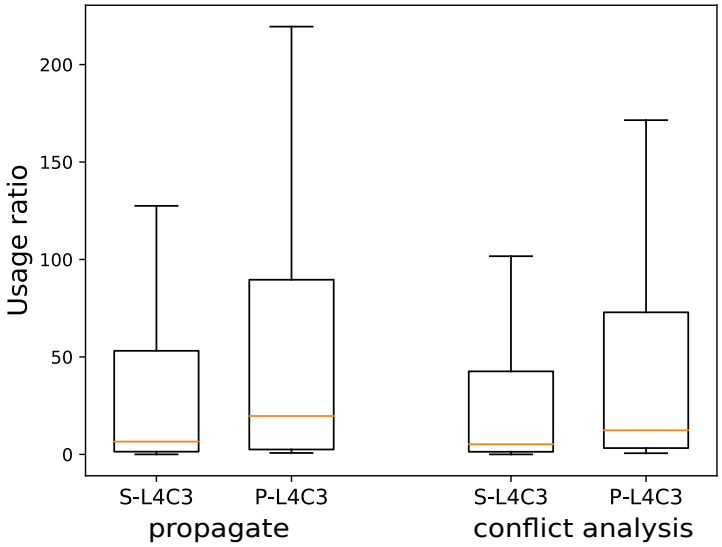


FIGURE 3.10 – Efficacité de notre stratégie de partage.

Les données collectées sont présentées dans les diagrammes de quartiles de la Figure 3.10 (la paire de gauche montre la propagation unitaire et la paire de droite l'analyse de conflit). Les diagrammes notés S-L4C3 représentent le ratio d'utilisation de l'ensemble de clauses correspondant dans MapleCOMSPS (le solveur séquentiel utilisé), tandis que ceux notés P-L4C3 font de même pour P-MCOMSPS-L4C3. Nous notons que lorsqu'un solveur ajoute une clause reçue à sa base de données, la LBD est celle d'origine. C'est bien cette valeur qui est indiquée dans les chiffres de P-MCOMSPS-L4C3.

Les clauses partagées dans P-MCOMSPS-L4C3, ont clairement un impact positif sur le comportement des moteurs séquentiels sous-jacents. Elles apportent de nouvelles informations utiles pour les procédures de propagation unitaire et d'analyse de conflit. Par exemple, en comparant le ratio d'utilisation dans la propagation unitaire des clauses dans S-L4C3 et P-L4C3,

nous voyons que le ratio de ces clauses va au-delà de 50 dans seulement $\approx 25\%$ des problèmes pour S-L4C3 et plafonne à 130. Dans P-L4C3, ce ratio va au-delà de 90 dans $\approx 25\%$ des problèmes et plafonne légèrement au-dessus de 200. La même observation vaut pour la procédure d'analyse de conflit. En outre, les médianes pour les diagrammes de l'approche parallèle sont toutes supérieures aux médianes correspondantes de l'approche séquentielle.

3.7 CONCLUSION ET PERSPECTIVES

La plupart des solveurs SAT parallèles utilisent des métriques de qualité locales (la plus pertinente étant la LBD) pour sélectionner les clauses apprises qui doivent être partagées. Dans ce chapitre, nous avons proposé de combiner cette métrique avec une métrique plus globale (COM) basée sur la structure de communauté des formules SAT. Le principe est d'utiliser le critère de communauté comme filtre pour un ensemble de clauses sélectionnées par la LBD, dans le but d'augmenter le ratio d'utilisation des clauses partagées.

Nous avons conçu un outil pour analyser les caractéristiques des clauses apprises dans un contexte séquentiel. À la suite de cette analyse, nous avons élaboré une politique de partage de clauses apprises, qui combine LBD et COM, dans un contexte parallèle. Nous avons démontré les qualités de cette stratégie en la mettant en œuvre dans `P-MCOMSPS` et montré qu'elle surpasse le solveur original sur les instances de la compétition SAT 2016, 2017, 2018 et 2019.

Nous avons en tête différentes façons d'améliorer ce travail. Tout d'abord, nous pouvons rechercher une approche plus dynamique dans notre méthode de filtrage. Cela permettrait de résoudre le problème que nous avons mentionné dans l'analyse du résultat de la Section 3.6.2. Nous avons noté que certains cas profitent d'un filtrage plus souple des clauses : nous pensons pouvoir utiliser un delta autour d'une certaine valeur de seuil sur la LBD, tout en maintenant notre seuil pour COM. Nous pouvons étudier le débit des clauses partagées requis pour différents types d'instances ainsi que le débit autorisé par différents types de matériel afin d'augmenter ou diminuer le LBD en conséquence. Il convient de noter que Hamadi et al. ont développé une idée similaire, mais en utilisant la taille comme métrique [44].

Deuxièmement, nous aimerions étudier l'effet de l'utilisation des communautés comme métrique de décision pour la réduction de la base de données de clauses apprises. Cette dernière est l'un des principaux composants d'un solveur SAT (voir Section 2.2.6), car le solveur peut apprendre des millions de clauses au cours de la résolution : conserver toutes ces clauses ralentit la propagation unitaire et entraîne des problèmes de mémoire. Ceci est amplifié dans le contexte parallèle où un solveur séquentiel a son propre ensemble de clauses apprises enrichi par d'autres solveurs. Dans `P-MCOMSPS`, les clauses sont supprimées en fonction de leurs LBD. Il est logique d'utiliser une métrique locale pour la logique d'un composant local. Cependant, les résultats encourageants présentés dans ce chapitre nous laissent penser que l'extension de l'utilisation des communautés à l'algorithme de réduction de la base de données améliorerait encore les moteurs séquentiels.

Chapitre

4

Intégration et évaluation d'une stratégie de minimisation de clauses

Contents

4.1 Introduction

Dans le chapitre précédent, nous avons abordé l'amélioration du partage d'informations grâce à une meilleure sélection des clauses. Notre stratégie n'agit pas sur la structure de la clause, mais elle en extrait simplement des informations supplémentaires. Ces informations sont basées sur l'affectation qui a dérivé la clause (la LBD), une valeur dynamique et sur la structure de la formule (la valeur COM), une valeur statique. Un autre critère structurel de qualité d'une

clause est sa taille. Naturellement, une clause de petite taille nécessite moins de décisions pour rentrer en conflit et donc élimine un plus large ensemble de l'espace de recherche. En pratique, cette mesure est liée à la LBD : plus une clause est petite, plus il y a de chance que sa LBD soit faible. Dans le contexte séquentiel, la LBD se révèle plus intéressante à exploiter comme critère de qualité. Cependant, agir sur ce critère à l'extérieur du solveur est compliqué en raison de sa localité. En revanche, comme nous l'avons vu dans la Section 2.2.7, il existe plusieurs techniques [46, 48, 80, 92] telles que la vivification permettant de simplifier la formule en réduisant la taille des clauses. Nous appelons ce processus *minimisation*. Nous pourrions utiliser ces techniques pour réduire la taille des clauses apprises partagées et ainsi améliorer la qualité du partage. Cependant, le problème de minimiser une clause de manière optimale est lui-même NP-difficile, car il nécessite de résoudre la formule [99]. On peut alors se poser les questions : sur quel fil d'exécution (ou *thread*) doit s'effectuer cette minimisation ? Doit-elle être effectuée par un des threads en charge de la résolution de la formule, par le thread en charge du partage, ou par un autre composant à part entière ? Wieringa et Heljanko [100] introduisent un composant découplé de la résolution (associé à son propre thread) afin d'effectuer cette résolution de manière asynchrone. Ce mécanisme est employé pour aider un solveur séquentiel.

La première contribution présentée dans ce chapitre est l'intégration de ce composant au sein d'un solveur parallèle qui permet de délivrer un flux constant de clauses minimisées aux travailleurs participant à la résolution. Ce mécanisme a prouvé son efficacité au sein du solveur `P-mcomsps` [98] lors des compétitions SAT 2020 et 2021 [1]. La politique de partage de `P-mcomsps` a été progressivement améliorée au fil des ans avec de multiples options, dont certaines ont précédé cette thèse, en plus de la minimisation asynchrone. Ces mécanismes ont prouvé leur intérêt au sein d'études expérimentales et de compétitions antérieures. Nous souhaitons mettre à jour nos intuitions en étudiant l'intégration progressive de ces mécanismes au sein d'un solveur SAT parallèle utilisant un algorithme séquentiel moderne.

Notre seconde contribution consiste donc en l'évaluation incrémentale de différents mécanismes afin de définir une stratégie de partage compétitive pour le solveur parallèle `Parkissat-RS`. Lors de la compétition SAT parallèle 2022, ce solveur, une instance du framework `Painless` (Section 2.3.4) utilisant `Kissat-MAB` [28] comme moteur séquentiel, a obtenu de très bons résultats tant sur les instances SAT que UNSAT. Les politiques de partage utilisées par les auteurs de `Parkissat-RS` sont beaucoup plus simplifiées par rapport à celles utilisées dans `P-mcomsps`. Nous voulons examiner ici si les politiques de partage de `P-mcomsps` peuvent améliorer les performances de `Parkissat-RS` [103]. Certaines de ces options sont de simples heuristiques, tandis que d'autres nécessitent un thread dédié. Puisque nous voulons utiliser un solveur avec un moteur séquentiel et un mécanisme de diversification très différent, nous devons à nouveau étudier les compromis entre la perte d'un cœur de calcul dédié à la résolution de la formule pour un cœur dédié à l'optimisation du partage.

Contributions.

— Nous présentons l'intégration d'un algorithme de minimisation au sein d'une architecture parallèle basée sur le solveur séquentiel `MapleCOMSPS`. Notre stratégie consiste à ajouter un flux de clauses réduites aux flux de clauses apprises partagées initialement au sein d'un solveur parallèle. L'algorithme de renforcement lui-même est tiré de [100]. Cette stratégie est évaluée au sein de différentes stratégies de partage et de parallélisation.

— Nous présentons l'évaluation de différentes configurations pour une politique de partage sur un solveur parallèle basée sur `Kissat-MAB`. Ces configurations sont basées sur les options suivantes :

1. Utilisation de plus de threads pour gérer le partage afin d'augmenter la bande passante des clauses apprises.
2. Utilisation d'un seuil dynamique pour la limite de la LBD.
3. Ajout d'un thread chargé de minimiser les clauses apprises partagées.
4. Mise en place d'un mécanisme pour éviter le partage de clauses apprises en double.

— Nous présentons également une étude de passage à l'échelle des configurations présentant les meilleurs résultats.

Structure du chapitre. La Section 4.2 présente les travaux précédents réalisés sur la minimisation, la mise en œuvre de l'algorithme de minimisation et son évaluation. La Section 4.3 présente la combinaison de la minimisation avec d'autres mécanismes visant à améliorer le partage dans un solveur parallèle moderne.

4.2 Partie 1 : La minimisation de clauses apprises rendue asynchrone

4.2.1 *Minimisation de clauses en inprocessing*

Dans la Section 2.2.7, nous avons examiné des méthodes de prétraitement efficaces permettant de simplifier la formule. Des efforts ont été déployés pour appliquer ces solutions pendant la résolution, c'est-à-dire *inprocessing*. Cette approche est particulièrement utile dans le cadre des solveurs CDCL, car ces solveurs peuvent apprendre de nouvelles contraintes dynamiquement (Section 2.2.3). Les clauses apprises peuvent également être simplifiées afin de renforcer leur utilité dans la propagation unitaire. De nombreux travaux ont été réalisés pour minimiser les clauses apprises dans un contexte séquentiel [66, 92, 93, 99]. Nous notons en particulier une méthode appelée *Learnt Clause Minimization (LCM)* [66], qui, comme l'approche que nous allons utiliser dans ce chapitre, repose largement sur la *vivification* présentée dans la Section 2.2.7. Lorsque la vivification est effectuée en *preprocessing*, les clauses générées dépendent uniquement des clauses de la formule d'origine (et des clauses générées par la vivification elle-même). En revanche, en inprocessing, la minimisation dépend également des

clauses apprises par le solveur. Afin de garantir que la minimisation reste indépendante de toute décision de branchement, LCM ne s'active qu'au niveau 0 de la résolution, c'est-à-dire qu'elle peut potentiellement s'activer à chaque redémarrage (voir Sections 2.2.2 et 2.2.4). Évidemment, afin de ne pas ralentir le solveur, LCM n'est pas activée à chaque redémarrage, mais selon un seuil ici défini en fonction du nombre de clauses apprises et de redémarrages depuis le dernier appel. De plus, LCM ne s'attarde pas à réduire toutes les clauses apprises. La métrique de sélection est la LBD de la clause, ce qui signifie que les clauses ayant une LBD élevée sont ignorées. LCM a prouvé son efficacité en étant déployée par des solveurs ayant remporté des compétitions SAT séquentielles (compétitions SAT de 2017 à 2019 [1]). Cependant, notre intérêt se porte sur l'exploitation des machines multicœurs pour effectuer des tâches qui sont a priori trop lourdes pour le contexte séquentiel. Dans ce contexte, il est moins nécessaire de se soucier du développement d'heuristiques pour déterminer quelles clauses réduire et quand, un fil d'exécution dédié à la minimisation peut simplement tourner sans interruption. Ainsi, nous nous inspirons d'une méthode, plus ancienne, qui intègre un composant rendant la minimisation des clauses apprises asynchrone pour un solveur séquentiel. Cette méthode décrit par Wieringa et Heljanko [100] appelle ce composant `Reducer`. La prochaine section donne un aperçu de cet article.

4.2.2 *Le `Reducer` : rendre la minimisation asynchrone*

L'algorithme de minimisation des clauses apprises lors de l'inprocessing est désormais dissocié de l'algorithme CDCL et s'exécute dans un fil d'exécution séparé. Le `Reducer` est le composant qui encapsule ce fil d'exécution et l'algorithme de minimisation. La communication entre le solveur séquentiel et son `Reducer` associé se fait grâce à deux structures de données en mémoire partagée, dont l'accès est synchronisé à l'aide d'un verrou.

La première structure, appelée *work set* est une file de taille bornée, avec une capacité de 1000 clauses dans le papier. C'est dans cette structure que le solveur place ses clauses apprises afin qu'elles puissent être minimisées. Bien qu'une architecture parallèle permette de traiter un nombre plus important de clauses que l'inprocessing séquentiel, les auteurs justifient le choix d'un *work set* par la volonté de donner la priorité aux clauses récentes. En effet, si une clause apprise est trop ancienne (la *durée* ici peut-être mesurée par le nombre de backjump ou redémarrage depuis son apprentissage) elle sera potentiellement moins utile au solveur, désormais situé dans un sous-espace de recherche complétement différent. De plus, lorsque le `Reducer` souhaite récupérer une clause du *work set* pour la réduire, il lui est fourni la *meilleure clause* présente. Le papier présente deux implémentations, une basée sur le solveur `MiniSat` [37], pour laquelle la meilleure clause est la plus petite et une autre basée sur le solveur `Glucose` [9] pour laquelle la meilleure clause est celle avec la LBD plus petite. Le `Reducer` récupère donc les clauses apprises depuis le *work set* et, s'il arrive à les réduire, les place dans la seconde structure de données, une simple file cette fois non limitée en taille.

Le pseudo-code de l'algorithme de minimisation utilisé par le `Reducer` que nous avons récupéré est présenté dans l' Algorithme 4. Les bases théoriques de cette technique sont présentées dans [100] et Section 2.2.7. Cette section se concentre uniquement sur les détails techniques.

```
1  function strengthening(C_in : clause) : la clause réduite
2      C_out := ∅
3      α := ∅
4      φ' := φ ∪ L_R
5      pour l ∈ (C_in \ C_out) s.t. ¬l ∉ propagationUnitaire(φ', α) faire
6          si ∅ ∈ φ'|_α alors
7              (L_R, C_new) := analyse(φ', α)
8              retourner C_new
9          C_out = C_out ∪ {l}
10         α = α ∪ {¬l}
11     L_R = L_R ∪ {C_out}
12     retourner C_out
```

Algorithme 4 : L'algorithme de minimisation

Algorithme 4 prend une clause C_{in} en entrée, et potentiellement produit une clause de taille réduite par rapport à C_{in} (Lignes 8 et 12). Il considère une affectation vide α (Ligne 3), toutes les clauses du problème φ et il gère son propre ensemble de clauses apprises L_R (Ligne 4) vide au début du programme.

Pour réaliser sa tâche, l'algorithme attribue itérativement la valeur $\bot$ à chaque littéral de la clause C_{in}, jusqu'à ce qu'il atteigne un conflit ou qu'il affecte avec succès tous les littéraux de la clause d'entrée. Il y a donc deux sorties possibles, respectivement C_{new} et C_{out}. À chaque itération, l'algorithme choisit un littéral dont le complémentaire n'est pas impliqué par l'affectation courante ($\neg\ell \notin$ `propagationUnitaire`(φ', α)). Cela garantit le dépouillement de la clause d'entrée de tous les littéraux qui sont impliqués par le reste de la clause. Ensuite, il exécute une propagation unitaire (Ligne 5). Si aucun conflit n'est découvert, le littéral est ajouté à la clause de sortie C_{out} et sa négation est ajoutée à l'ensemble d'affectation α (Lignes 9 et 10).

Lorsqu'un conflit est atteint (Ligne 6), la fonction `analyse()` est alors appelée (Ligne 7) : elle exécute une séquence d'analyse de conflit, de *backjump* et de propagation unitaire jusqu'à la sortie du conflit ou avoir vidé l'ensemble α. Pendant cette phase, l'algorithme peut apprendre de nouvelles clauses (qui sont ajoutées à L_R). Lorsque `analyse()` atteint une zone sans conflit (tout en supposant α), il génère la clause C_{new} qui est retournée (Ligne 8). Cette dernière est composée de l'ensemble des littéraux : $\{\ell | \neg\ell \in \alpha\} \cup \{k\}$, k étant un littéral de $C_{out} \notin \alpha$.

Si tous les littéraux de C_{in} sont assignés avec succès, alors la clause C_{out} est ajoutée à L_R et ensuite retournée (Lignes 11 et 12).

4.2.3 *Intégration du `Reducer` dans Painless*

Le `Reducer` a été créé dans le but d'améliorer l'efficacité d'un solveur séquentiel. Notre objectif est d'augmenter la capacité de résolution d'un solveur parallèle en intégrant le `Reducer`

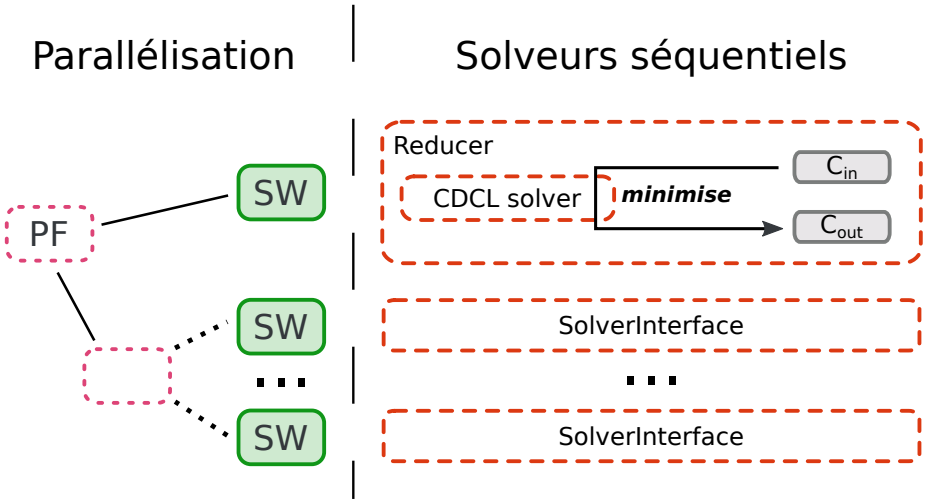


FIGURE 4.1 – Intégration du `Reducer` dans une stratégie de parallélisation arborescente

à une instance de Painless (Section 2.3.4). Ainsi, nous tirons parti de la modularité de la plateforme pour ajouter cet algorithme et bénéficier des mécanismes de parallélisation et de partage existants.

Intégration du `Reducer` en tant que moteur séquentiel. Nous décidons de faire du `Reducer` un type spécial de solveur séquentiel. Par conséquent, il implémente la `SolverInterface`, bien que seules les méthodes suivantes soient implémentées : `solve`, `addLearnedClauses`, `getLearnedClauses`. La méthode `solve` consiste en une boucle infinie qui suit les étapes suivantes :

— Récupère les clauses apprises reçues avec `getLearnedClauses`.

— Pour chaque clause, crée la clause complémentaire et applique l'Algorithme 4. Cette opération utilise une une instance de `MapleCOMSPS`. L'algorithme séquentiel utilisé pour la minimisation est distinct de celui utilisé par les travailleurs chargés de résoudre la formule, bien que dans l'implémentation présentée dans cette section, les travailleurs sont également des instances de `MapleCOMSPS`.

— Si la clause est effectivement réduite, elle est préparée pour la distribution par l'appel à `addLearnedClauses`. Si la clause a été réduite à la clause vide, cela signifie que la formule est insatisfiable (UNSAT) et le `Reducer` termine la résolution en renvoyant ce résultat. Si le `Reducer` peut retourner UNSAT, il ne peut pas trouver de solution si la formule est satisfiable (SAT).

Intégration du `Reducer` dans la stratégie de parallélisation. Faire du `Reducer` un algorithme exécuté par un thread dédié est simple, il suffit d'ajouter le composant à une stratégie de parallélisation qui lui attribue un `SequentialWorker`. Cependant, le `Reducer` n'étant pas un algorithme complet de résolution (il n'implémente pas l'intégralité des méthodes de la

`SolverInterface`), il n'aurait pas de sens d'inclure ce composant dans une stratégie de parallélisation complexe telle que le divide-and-conquer par exemple. Ainsi, si la stratégie de parallélisation voulue nécessite une structure arborescente, il sera nécessaire d'intégrer cette stratégie à la racine d'un portfolio au côté du `Reducer`. La Figure 4.1 présente un exemple de configuration où le `Reducer` est intégré à un portfolio, en conjonction avec une stratégie de parallélisation arborescente quelconque (comme le divide-and-conquer).

Intégration du `Reducer` dans la stratégie de partage. Nous ne récupérons pas les structures de données utilisées dans l'article présenté pour la communication. Nous profitons des structures *lockfree* déjà implémentées dans Painless pour distribuer les clauses. L'enchaînement des étapes permettant la minimisation asynchrone dans notre architecture est le suivant :

— Comme présenté dans la Section 2.3.4, les solveurs séquentiels ajoutent leurs clauses apprises dans un *buffer d'export* (selon le filtre de clauses appliqué).

— Le `Sharer` récupère les clauses des *buffers d'export* des solveurs producteurs (c'est-à-dire tous les solveurs, `Reducer` compris, dans notre cas) et les place dans les *buffer d'import* des consommateurs. De nouveau, ce groupe comprend tous les solveurs.

— Le `Reducer` récupère les clauses apprises de son *buffer d'import* et peut donc exécuter l'Algorithme 4 afin de tenter de les réduire.

— Les clauses effectivement réduites sont placées dans le *buffer d'export* du `Reducer`.

L'avantage de cette implémentation est qu'elle repose sur une structure *lockfree*. Cependant, il n'y a pas de limite sur la taille du *buffer d'import* du `Reducer` et aucun tri n'est effectué pour permettre au `Reducer` de récupérer la *meilleure clause*. Ce dernier point ne devrait pas poser problème, car la stratégie de partage ne devrait partager que des *clauses de qualité*. À l'inverse, le premier point pourrait être amélioré dans une future implémentation.

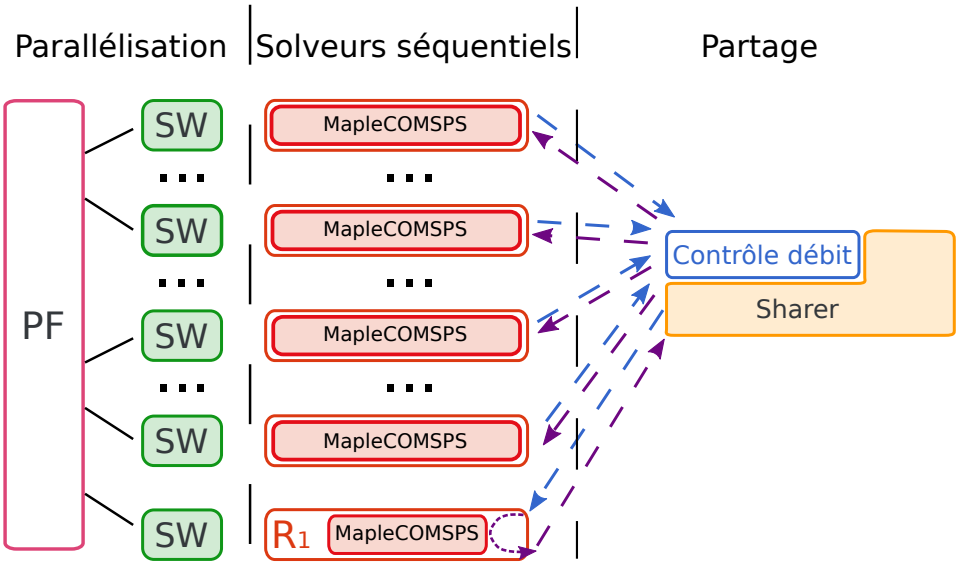


FIGURE 4.2 – Architecture of `P-mcomsps`

La Figure 4.2 présente une instance de `Painless` similaire au `P-mcomsps` déjà présentée, mais avec un `Reducer` intégré au portfolio (R_1 sur la figure).

4.2.4 *Évaluation du `Reducer` sur différentes stratégies de parallélisation*

Pour évaluer les performances du composant développé et étudier son impact dans différents solveurs parallèles, nous avons intégré un `Reducer` dans plusieurs stratégies de parallélisation. Nous avons ensuite réalisé un ensemble d'expériences pour comparer les résultats.

Description des solveurs. Tous les solveurs parallèles que nous avons construits, sauf un, sont basés sur `P-MCOMSPS` [59]. Il implémente une stratégie de type Portfolio (PF, Section 2.3.2) et utilise `MapleCOMSPS` [62] comme moteur séquentiel. Les solveurs diffèrent cependant par leurs stratégies de partage, qui sont toutes des exploitations différentes de la LBD des clauses apprises.

Nous avons donc dérivé les stratégies suivantes :

— **horde** : L'option `-horde`, extraite du solveur `HordeSat` [11], ajuste le partage des clauses lorsque le `Sharer` estime qu'un producteur n'envoie pas assez de clauses. Par conséquent, le `Sharer` peut dynamiquement augmenter le seuil de LBD d'un producteur particulier. Au début de la résolution, ce seuil est fixé à 2.

— **L**i partage uniquement les clauses apprises ayant une valeur de LBD $\leq$ i.

Ainsi, nous avons fini par développer le solveur `P-MCOMSPS-horde` (la stratégie utilisée par le vainqueur de la piste parallèle de la compétition SAT 2018), le solveur `P-MCOMSPS-L2` (L2 la stratégie utilisée par le second de la piste parallèle de la compétition SAT 2018.) et le solveur `P-MCOMSPS-L4` (L4 s'est montré performante sur des études expérimentales précédentes, voir la Table 3.4 et la Figure 3.3).

Pour compléter notre expérimentation, nous avons également développé un solveur de type Divide-and-Conquer (DC, Section 2.3.1) qui utilise la stratégie de partage L4. Nous appelons ce solveur `DC-MCOMSPS-L4` [58].

TABLE 4.3 – Résultats des différents solveurs sur le benchmark SAT 2018

Parallélisation	Solveurs	PAR-2	CTI	UNSAT	SAT	PROBLÈMES(400)
PF	P-MCOMSPS-L4	363h06	26h53	115	165	280
	P-MCOMSPS-L4-str	**342h33**	**21h47**	**121**	**168**	**289**
	P-MCOMSPS-L2	379h32	23h04	108	**165**	273
	P-MCOMSPS-L2-str	**371h53**	**20h45**	**115**	163	**278**
	P-MCOMSPS-horde	356h13	37h10	121	165	286
	P-MCOMSPS-horde-str	**342h36**	**32h15**	**125**	**167**	**292**
DC	DC-MCOMSPS-L4	448h34	**17h17**	100	146	246
	DC-MCOMSPS-L4-str	**437h44**	18h59	**103**	**149**	**252**

Pour chacun de ces solveurs, nous avons créé son homologue comprenant le composant `Reducer`. Nous les avons appelés en étendant leurs noms par `-str` (par exemple, `P-MCOMSPS-L4-str`). Il est important de noter que nous n'utilisons pas un cœur supplémentaire pour le `Reducer`, par exemple, si nous utilisons 12 cœurs pour le `P-MCOMSPS-L4`, nous utilisons également 12 cœurs pour le `P-MCOMSPS-L4-str` (un thread effectue la minimisation à la place de l'algorithme CDCL).

Résultats expérimentaux. Pour l'évaluation, nous utilisons le benchmark principal de la compétition SAT 2018 [1] qui contient 400 instances. Tous les travaux ont été exécutés sur un processeur Intel Xeon à 2,40 GHz et 1,48 To de RAM. Les solveurs ont été lancés avec 12 threads, une limite de mémoire de 150 Go et un timeout de 5000 secondes (le timeout est le même que pour les compétitions SAT).

Les performances de nos solveurs sont évaluées à l'aide des métriques suivantes :

— le temps d'exécution moyen pénalisé (PAR-2) additionne le temps d'exécution d'un solveur et pénalise les exécutions qui dépassent le timeout d'un facteur 2

— Le nombre de problèmes résolus par un solveur

— *cumulative time of the intersection* (CTI) additionne le temps d'exécution d'un solveur sur les problèmes résolus par tous les solveurs.

La Table 4.3 présente les résultats de nos expériences, où chaque solveur est comparé à son homologue (avec un composant `Reducer`). Les cellules ombragées indiquent lequel des deux solveurs a les meilleurs résultats. Nous observons que dans toutes les métriques, sauf dans deux cas, les versions avec un `Reducer` sont meilleures : plus d'instances sont résolues et de meilleures valeurs PAR-2 sont obtenues dans tous les cas. Seul le CTI de la version DC n'est pas aussi bon que les autres valeurs. De même, les gains en nombre d'instances résolues semblent plus importants dans la catégorie UNSAT, mais le nombre d'instances SAT s'améliore également.

Les Figures 4.4 et 4.5 présente les performances des solveurs par instance, séparés selon que les instances sont SAT ou UNSAT. Les axes de ces figures représentent le temps d'exécution en secondes. Les points situés au-delà des lignes pointillées, au-dessus du niveau de 5000 secondes, représentent un *timeout*. Ces figures montrent clairement l'apport du `Reducer` sur les instances UNSAT, mais confirme que son apport est moins marqué sur les instances SAT.

1. http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip

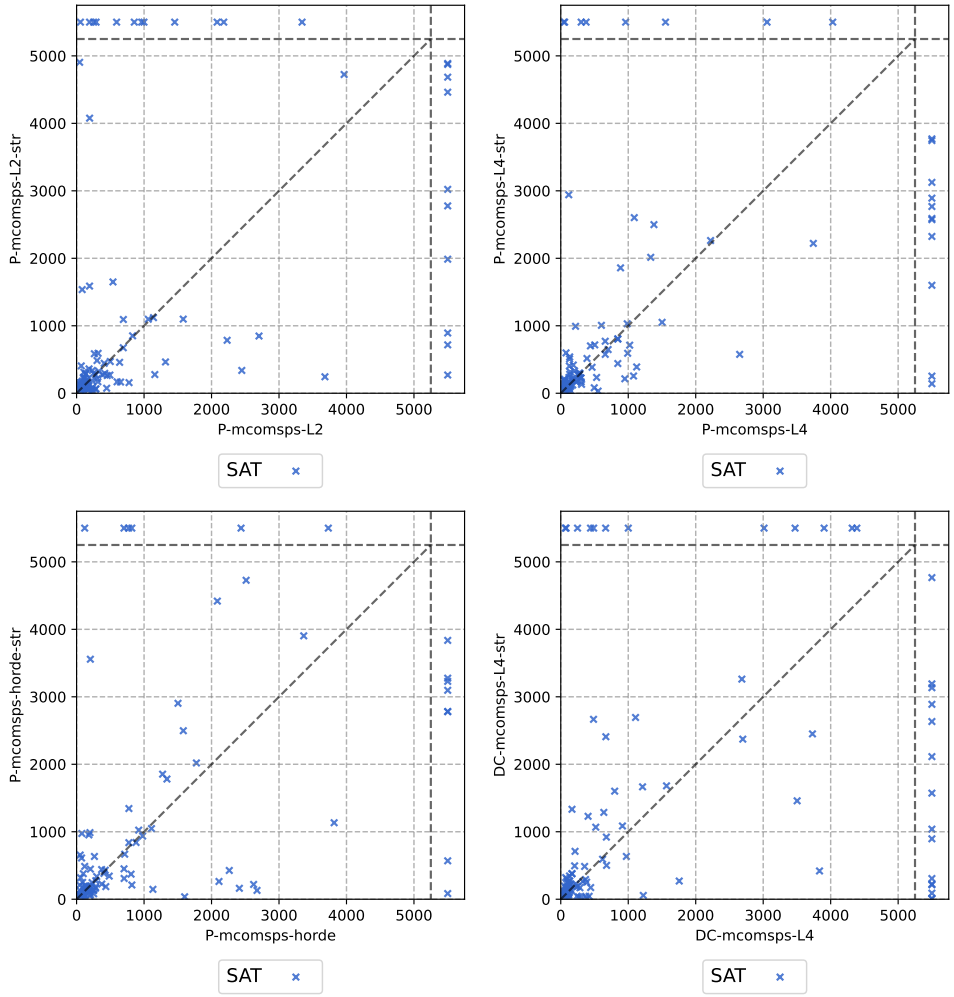


FIGURE 4.4 – Evaluation par instance SAT du mécanisme `-str` sur le benchmark SAT 2018

Pour aller plus loin dans notre évaluation, nous avons mesuré les capacités de minimisation du `Reducer` sur les instances que chaque solveur a pu effectivement résoudre, en écartant celles où le `Reducer` n'a reçu aucune clause (problème résolu trop rapidement) :

- `P-MCOMSPS-L4-str` (255 instances), 44,21% des clauses traitées par le `Reducer` sont effectivement raccourcies. La taille moyenne de ces clauses après renforcement est inférieure de 25,45% à la moyenne de leur taille originale ;
- `P-MCOMSPS-L2-str` (257 instances), a traité 32,59% des clauses et a réduit leur taille de 23,67% ;
- `P-MCOMSPS-horde-str` (258 instances) a traité 34,79% des clauses et a réduit leur taille de 27,75%
- `DC-MCOMSPS-L4-str` (245 instances) a réduit de 18,86% 28,80% des clauses.

En conclusion, le `Reducer` a réussi à réduire $1/3$ des clauses qu'il reçoit de $1/4$ de leur taille.

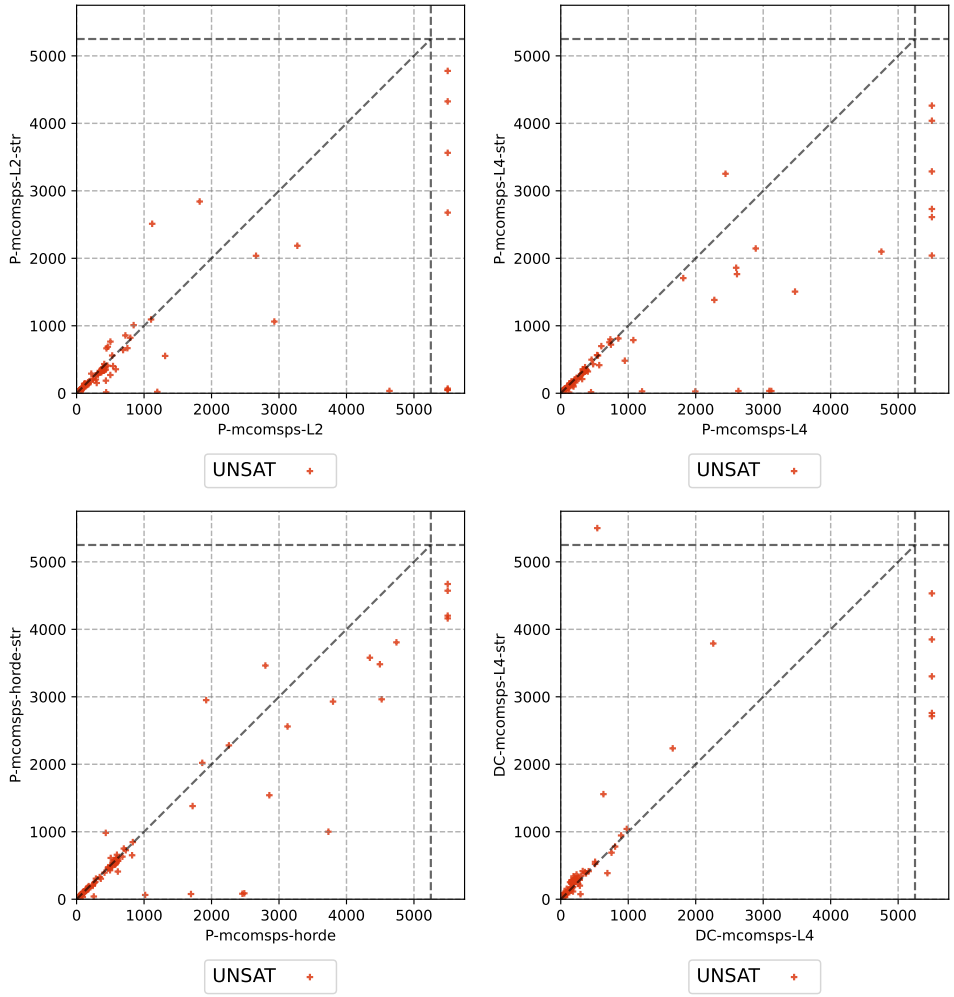


FIGURE 4.5 – Evaluation par instance SAT du mécanisme `-str` sur le benchmark SAT 2018

4.3 PARTIE 2 : POLITIQUES DE PARTAGE OPTIMISÉES POUR SOLVEURS SAT PARALLÈLES

Dans cette section, nous améliorons `Parkissat-RS` [103], un solveur parallèle (portfolio) utilisant une communication basée sur la mémoire partagée entre les solveurs séquentiels sous-jacents. En effet, en tant que vainqueur de la compétition SAT parallèle 2022 ce portfolio personnalisé de solveurs `kissat` [28] s'est avéré très efficace. Ce solveur est une instance de `Painless` et sa configuration est globalement la même que `P-mcomsps` décrit dans la Section 2.3.4.

Les différences se situent dans le mécanisme de diversification d'une part. `Parkissat-RS` est basé sur la randomisation de l'ordre de branchement des variables pour chaque solveur du portfolio, complété par les options de diversification de `kissat` (le solveur sous-jacent utilisé). D'autre part, le filtre de sélection de clauses à partager n'est pas dynamique. En effet,

`Parkissat-RS` n'utilise pas la stratégie dynamique de `HordeSat`, ainsi le solveur partage uniquement les clauses apprises avec un LBD ≤ 2.

Dans les Sections 4.3.2 à 4.3.5, nous ajouterons progressivement différents mécanismes à la politique de partage de l'architecture décrite ci-dessus et étudierons leurs effets sur les performances du solveur.

4.3.1 *Conception expérimentale et méthodologie d'évaluation*

Nous présentons ici la méthodologie utilisée pour évaluer les combinaisons possibles d'options et détaillons la plateforme expérimentale.

Le benchmark Nous lançons nos solveurs sur les instances de la *main track* de la compétition SAT 2022. Ce benchmark contient au moins 171 instances SAT et 187 instances UNSAT selon les résultats de la compétition. Nous séparons donc les résultats SAT et les résultats UNSAT en deux tableaux. Les 42 instances dont les résultats sont UNKNOWN selon la compétition (et nos expériences) sont éliminées des tableaux.

Évaluation des performances Nous voulons voir comment chaque solveur se compare au *Virtual Best Solver (VBS)*, un solveur composé des meilleurs résultats pour chaque instance, en termes de temps d'exécution et d'instances résolues. Pour chaque tableau, nous confrontons les performances d'un solveur avec le nombre de cœurs dédiés à l'algorithme CDCL. En effet, nous souhaitons utiliser tous les cœurs physiques disponibles sur nos machines de test, alors que certaines de nos options nécessitent un thread dédié pour les exécuter. Par conséquent, dans certaines configurations, un cœur physique dédié à l'algorithme CDCL (`Parkissat-RS` dans la version originale) doit être sacrifié en faveur de l'option. Voici les données représentées dans les différents tableaux :

— La colonne CDCL indique le nombre de cœurs CDCL utilisés dans le portfolio.

— Le temps d'exécution moyen pénalisé (PAR-2) : additionne le temps d'exécution d'un solveur et pénalise les exécutions qui dépassent le timeout d'un facteur 2

— Nombre d'instances résolues.

Le matériel Les expériences ont été réalisées sur un cluster Intel Xeon Silver 4216 avec 32 cœurs et 384 Go de RAM. Les solveurs ont fonctionné avec un délai d'attente de 5000s et une limite de mémoire de 256Go.

Dans les prochaines sections, nous présenterons diverses améliorations de la politique de partage de `Parkissat-RS` et étudierons leurs performances. Chaque mécanisme est étudié individuellement, mais aussi en combinaison avec les autres. En effet, ils ont des effets sur le partage des clauses, à la fois quantitatifs et qualitatifs. De plus, ils ont également des coûts en

termes de ressources dédiées. Il est donc important d'étudier les effets secondaires de l'un sur l'autre.

Afin de simplifier la présentation des résultats, nous avons choisi d'ajouter chaque mécanisme étudié étape par étape. Le nom du solveur résultant est `P-[options]`.

4.3.2 *XG : Augmenter le débit partagé en utilisant plusieurs groupes de production*

La première option que nous ajoutons à la politique met en œuvre l'idée de *multiples groupes de production*. Un groupe de production se compose d'un ensemble de solveurs en tant que producteurs, de tous les solveurs du portfolio en tant que consommateurs et d'un thread `Sharer` de partage fonctionnant sur un cœur physique dédié.

L'implémentation originale comporte un seul groupe de production (dans les tableaux, cette implémentation est appelée `P-1G`). Pour augmenter le débit des clauses partagées, nous introduisons un deuxième groupe de production : la moitié des solveurs du portfolio sont des producteurs d'un groupe et l'autre moitié sont des producteurs de l'autre groupe. Cette nouvelle configuration est appelée `P-2G`.

Le mécanisme proposé est entièrement personnalisable et rien n'empêche les groupes de se multiplier. Cependant, des tests préliminaires ont montré que, sur notre machine de test, les performances sont dégradées lorsque plus de 2 groupes sont utilisés. En effet, le nombre de cœurs physiques dédiés aux différents threads devient trop important.

La Table 4.6 compare `P-1G` avec `P-2G`. Nous observons que l'utilisation de deux groupes de partage permet d'obtenir de meilleures performances sur les instances SAT, avec trois instances supplémentaires résolues. Cependant, cela n'améliore pas les performances sur les instances UNSAT. `P-2G` perd une instance, mais étant donné le temps d'exécution presque équivalent, `P-1G` ne semble pas particulièrement plus rapide. La Figure 4.7 confirme que l'option n'a pas d'effet sur les instances UNSAT.

L'ajout du second `Sharer` permet un partage plus rapide des clauses et donc une utilisation beaucoup plus rapide de ces clauses partagées. Cependant, comme la stratégie ne partage que les clauses dont la LBD est inférieur ou égal à deux, l'impact de ces clauses partagées est limité.

Solveurs	CDCL	PAR2	SAT
VBS	-	23H10	167
P-2G	30	**25H13**	**167**
P-1G	31	32H30	164

Solveurs	CDCL	PAR2	UNSAT
VBS	-	84H06	165
P-1G	31	**85H15**	**165**
P-2G	30	86H43	164

TABLE 4.6 – Évaluation de la performance de l'option `-XG`.

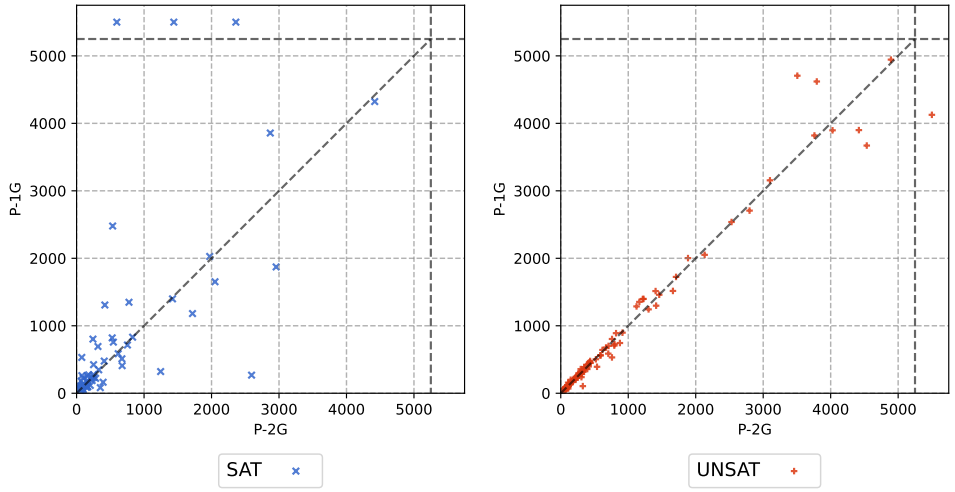


FIGURE 4.7 – Evaluation par instance de l'option `-XG`

4.3.3 *Horde : Une heuristique pour sélectionner les clauses à partager*

L'option `-horde` ajuste le partage des clauses lorsque le `Sharer` estime qu'un producteur envoie trop ou pas assez de clauses. Par conséquent, le `Sharer` peut dynamiquement augmenter ou diminuer le seuil de LBD d'un producteur particulier.

La Table 4.8 montre les performances de l'activation de `-horde` sur les configurations `P-1G` et `P-2G`. Bien que cette option semble dégrader légèrement les performances pour les instances SAT, en particulier `P-1G-horde` perd deux instances par rapport à `P-1G`, on peut observer sur la Figure 4.9 que sur les instances résolues par les deux solveurs, la version activant `-horde` est globalement plus rapide.

D'autre part, cette option permet un gain de performance important pour les instances UNSAT, avec jusqu'à cinq instances supplémentaires pour les deux solveurs avec cette option. La supériorité de l'option `-horde` sur ces instances se confirme également pour les instances résolues par les deux solveurs.

Il ressort de cette expérience que pour un grand nombre d'instances UNSAT, les clauses dont la LBD est supérieur à deux sont très utiles. En effet, l'option `-horde` est aussi performante que le VBS en termes d'instances UNSAT résolues.

Solveurs	CDCL	PAR2	SAT
VBS	-	17H36	169
P-2G	30	**25H13**	**167**
P-2G-horde	30	25H55	**167**
P-1G	31	32H30	164
P-1G-horde	31	35H21	162

Solveurs	CDCL	PAR2	UNSAT
VBS	-	69H44	169
P-1G-horde	31	**70H42**	**169**
P-2G-horde	30	71H03	**169**
P-1G	31	85H15	165
P-2G	30	86H43	164

TABLE 4.8 – Évaluation de la performance de l'option `-horde`.

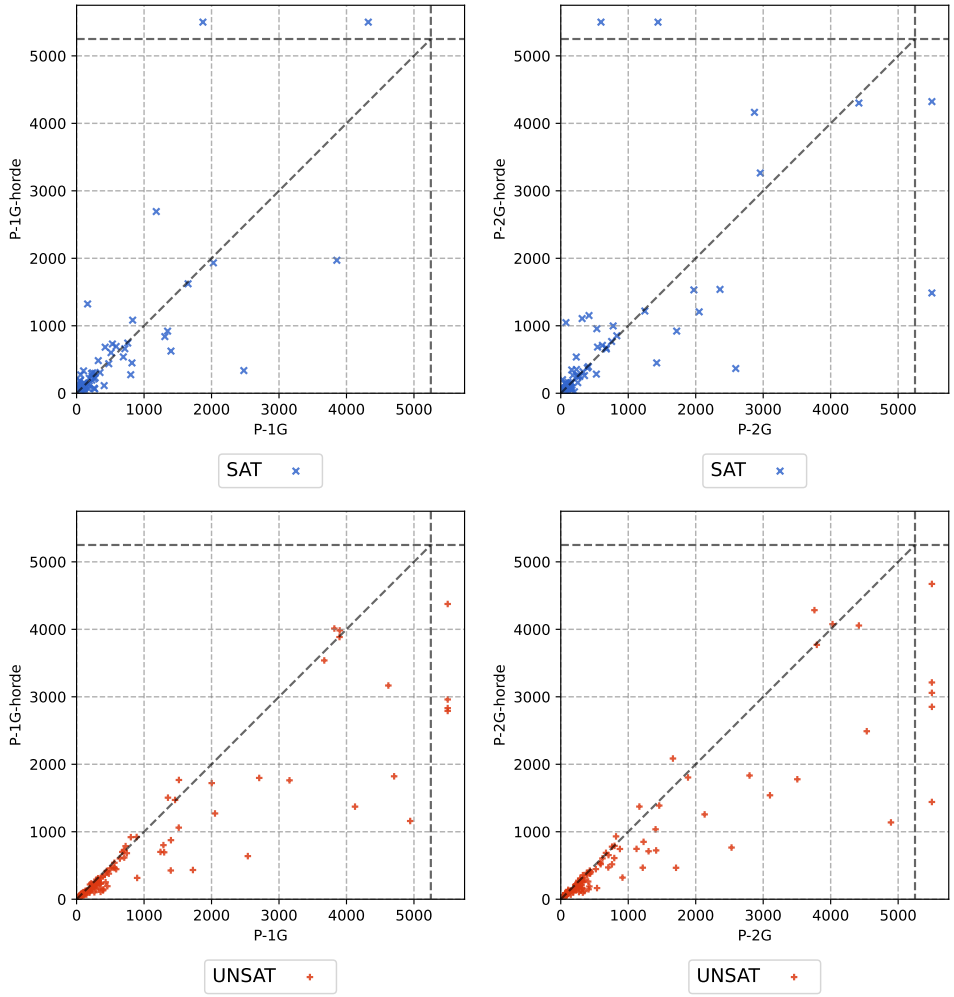


FIGURE 4.9 – Evaluation par instance de l'option `-horde`

4.3.4 *STR : Minimisation asynchrone des clauses*

L'option `-str` active un `Reducer` mettant en œuvre l'algorithme de minimisation présenté dans ce chapitre. Un `Reducer` est ajouté pour chaque groupe de partage, ainsi si l'option `-str` est activée avec l'option `-2G`, deux threads sont utilisés pour deux `Reducer`. Dans ce cas, le nombre de thread dédié à la résolution est réduit à 28.

La Table 4.10 montre les performances de l'option `-str`. On constate que les résultats de l'intégration de cette option sont stables par rapport aux instances UNSAT. En effet, elle améliore toutes les configurations déjà définies, soit par le temps d'exécution, soit par le nombre d'instances résolues. Ce dernier cas apparaît pour la configuration `P-2G-str` par rapport à la configuration `P-2G`. La Figure 4.11 montre les performances sur les instances UNSAT de l'option `-str` sur `P-1G-horde` et `P-2G-horde` (les deux meilleurs solveurs de cette expérience) et `P-2G` le solveur avec la différence la plus marquée. Les résultats sur `P-2G-str` sont particulièrement intéressants, car ils montrent que l'option `-str` permet de

Solveurs	CDCL	PAR2	SAT
VBS	-	16H22	169
P-2G	30	**25H13**	**167**
P-2G-horde	30	25H55	**167**
P-1G-horde-str	30	29H34	165
P-2G-str	28	31H26	164
P-1G	31	32H30	164
P-2G-horde-str	28	33H45	164
P-1G-str	30	34H33	163
P-1G-horde	31	35H21	162

Solveurs	CDCL	PAR2	UNSAT
VBS	-	67H39	169
P-2G-horde-str	28	**70H13**	**169**
P-1G-horde-str	30	70H14	**169**
P-1G-horde	31	70H42	**169**
P-2G-horde	30	71H03	**169**
P-2G-str	28	73H43	**169**
P-1G-str	30	85H04	164
P-1G	31	85H15	165
P-2G	30	86H43	164

TABLE 4.10 – Évaluation de la performance de l'option `-str`.

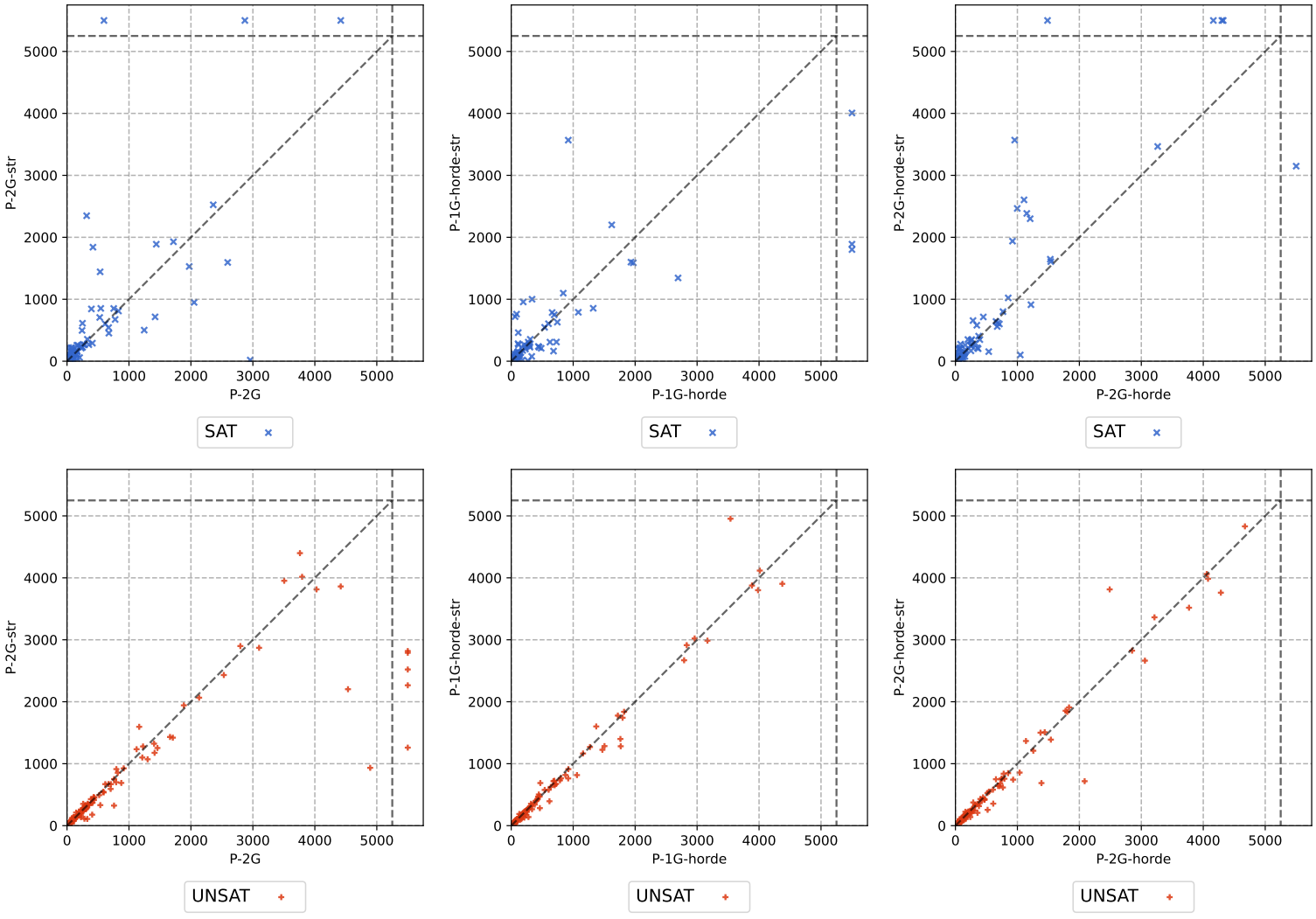


FIGURE 4.11 – Evaluation par instance de l'option `-str`

résoudre plus d'instances, mais contrairement à l'option `-horde`, n'accélère pas fortement les performances pour les instances résolues par les deux solveurs. Pour `P-1G-horde` et `P-2G-horde`, la différence est minime, mais globalement en faveur de l'activation de `-str`.

Pour les instances SAT, nous observons globalement que trois configurations sur quatre voient leurs performances dégradées par l'option. `P-1G-horde-str` semble être une anomalie qui pourrait être expliquée par une forme d'aléa qui affecte le comportement du solveur. La Figure 4.11 montre les performances sur les instances SAT de l'option `-str` sur `P-2G` et `P-2G-horde` (les deux meilleurs solveurs) et `P-1G-horde`, le solveur exposant une anomalie. Nous remarquons que pour les solveurs `P-2G` et `P-1G-horde` l'option `-str` ne diminue pas les performances pour les instances résolues par les deux solveurs, bien qu'elle n'arrive pas à trouver une solution pour certaines instances. Au contraire, combinée avec `P-2G-horde`, elle semble pénalisée le solveur dans tous les cas.

La conclusion de cette expérience est que pour les instances UNSAT, l'utilisation d'un cœur physique initialement dédié à l'exécution d'un CDCL pour exécuter un `Reducer` est bénéfique pour le solveur, bien que la différence sur les instances UNSAT est bien moins marquée que dans la Table 4.3. L'inverse est vrai pour les instances SAT.

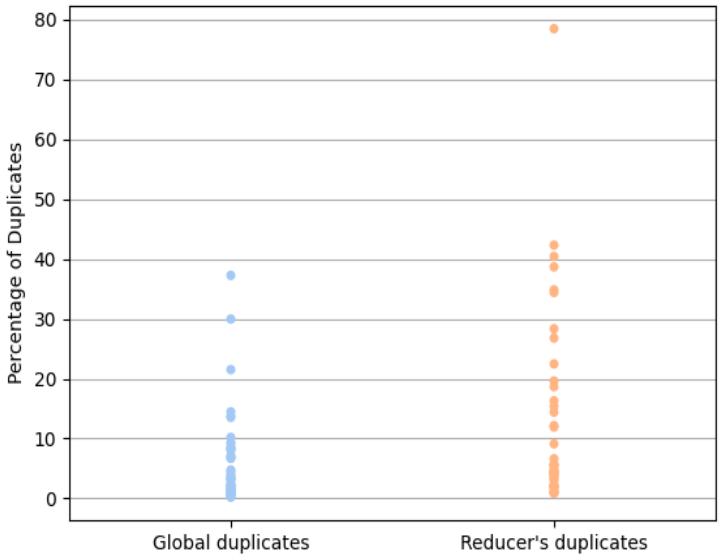


FIGURE 4.12 – Pourcentage de clauses dupliquées provenant des solveurs CDCL et d'un `Reducer`

4.3.5 *DUP : une option pour empêcher le partage de doublons*

Dans un contexte parallèle, il est très probable qu'il y ait de nombreuses clauses identiques apprises par les différents solveurs du CDCL. Par conséquent, le partage de ces doublons pourrait être préjudiciable à un solveur parallèle.

Une bonne idée pourrait être de développer un mécanisme permettant de détecter et de rejeter le partage de ces clauses dupliquées. Un tel mécanisme a un coût qui peut gêner le `Sharer`, nous devons donc nous assurer que le nombre de ces clauses est suffisamment important pour que le compromis en vaille la peine.

La Figure 4.12 présente une étude qui détermine le pourcentage de doublons provenant des solveurs CDCL (doublons globaux) et du `Reducer` (après la phase de minimisation). Chaque point montre le pourcentage pour une instance du benchmark [2].

Nous remarquons que les solveurs CDCL produisent principalement entre 0 et 10% de doublons. De plus, le `Reducer` produit un grand nombre de doublons et sur un plus grand nombre d'instances du benchmark. Cette analyse nous motive à introduire un mécanisme de suppression des doublons.

Pour mettre en œuvre un tel mécanisme, nous devons garder une trace des clauses déjà partagées. Un stockage parfait serait ingérable du point de vue de la mémoire. Nous avons donc décidé de mettre en œuvre une approche probabiliste peu encombrante basée sur les filtres de Bloom [23]. Le `Sharer` est étendu par un filtre de Bloom qui lui permet d'éviter de partager des doublons au sein de son groupe de production. Nous notons que cette idée est déjà présente dans `HordeSat`. Ce mécanisme est activé par l'option `-dup`.

La Table 4.13 présente la performance de l'option `-dup`. L'option semble dégrader les performances des instances SAT sur toutes les options. Sur les instances UNSAT la présence de l'option n'a aucun effet sur le nombre d'instances résolues et ne fait pas évoluer le temps de résolution de manière significative. La Figure 4.14 confirme pour une sélection de solveurs un comportement erratique sur les instances SAT et la stabilité sur les instances UNSAT.

Notre intuition suggère que si un filtre de Bloom est efficace, le processus de vérification des clauses dupliquées dans chaque phrase peut encore entraver le partage. Ce compromis ne semble pas utile, en particulier dans le contexte de ce benchmark particulier avec le solveur choisi.

4.3.6 *Étude de mise à l'échelle*

Jusqu'à présent, nos recherches se sont concentrées sur l'analyse de l'impact de différents mécanismes sur une configuration composée de 32 cœurs physiques, ce qui reflète les caractéristiques des machines utilisées lors des compétitions SAT ces dernières années. Cependant, la question de la mise à l'échelle de ces algorithmes est devenue une préoccupation majeure.

2. ici nous avons utilisé le benchmark de la compétition SAT 2021

Pour répondre à cette préoccupation, nous avons mené une nouvelle étude en utilisant deux architectures supplémentaires : l'une avec 48 cœurs et l'autre avec 64 cœurs. Dans le cadre de cette étude, nous nous sommes concentrés sur les combinaisons les plus performantes identifiées dans l'étude à 32 cœurs, à savoir `2G`, `2G-horde`, `2G-horde-str` et `1G-horde-dup`. Pour évaluer leurs performances, nous avons utilisé une batterie de tests comprenant 40 instances, résolues par au moins une configuration utilisant l'architecture à 32 cœurs. Les instances choisies sont parmi les plus complexes du benchmark, sur la base de leurs temps de résolution lors des expériences sur la machine à 32 cœurs. Chaque instance dispose d'un temps de résolution maximal autorisé, toujours de 5000 secondes. Si un solveur ne parvient pas à résoudre une instance dans le temps imparti, on ajoute simplement 5000 secondes à sa durée totale d'exécution (pas de pénalité comme le PAR-2). Les résultats de cette étude sont présentés dans la Figure 4.15.

Dans l'ensemble, nos résultats indiquent que la performance relative des différentes combinaisons, à l'exception d'un cas, reste constante lorsque le nombre de cœurs augmente, que le problème soit SAT ou UNSAT. Cependant, nous avons observé un phénomène intrigant : au-delà de 48 cœurs, l'augmentation des performances diminue notablement pour les problèmes UNSAT, quel que soit l'algorithme utilisé. Cela peut être attribué aux limitations des mécanismes de partage, car les threads supplémentaires ne réduisent pas efficacement l'espace de recherche exploré par les autres threads. En revanche, les problèmes SAT présentent un comportement différent, car l'augmentation du nombre de threads améliore la probabilité de trouver rapidement une solution.

En plus de ces résultats généraux qui confirment les recherches précédentes, il convient de

Solvers	CDCL	TIME	SAT
VBS	-	12H25	170
P-2G	30	25H13	167
P-2G-horde	30	25H55	167
P-1G-horde-str	30	29H34	165
P-2G-horde-str-dup	28	30H45	164
P-2G-str	28	31H26	164
P-1G	31	32H30	164
P-1G-str-dup	30	32H44	164
P-1G-dup	30	33H38	164
P-2G-horde-str	28	33H45	164
P-1G-horde-dup	31	34H22	163
P-1G-str	30	34H33	163
P-2G-horde-dup	30	35H01	163
P-1G-horde	31	35H21	162
P-2G-dup	30	37H41	162
P-1G-horde-str-dup	30	39H09	161

Solvers	CDDL	PAR2	UNSAT
VBS	-	67H03	169
P-1G-horde-dup	31	70H05	169
P-2G-horde-str	28	70H13	169
P-1G-horde-str	30	70H14	169
P-2G-horde-dup	30	70H31	169
P-1G-horde	31	70H42	169
P-2G-horde-str-dup	28	70H47	169
P-2G-horde	30	71H03	169
P-1G-horde-str-dup	30	72H19	168
P-2G-str	28	73H43	169
P-1G-dup	31	84H08	165
P-2G-dup	30	85H02	165
P-1G-str	30	85H04	164
P-1G	31	85H15	165
P-1G-str-dup	30	85H32	164
P-2G	30	86H43	164

TABLE 4.13 – Performance evaluation of `-dup` option

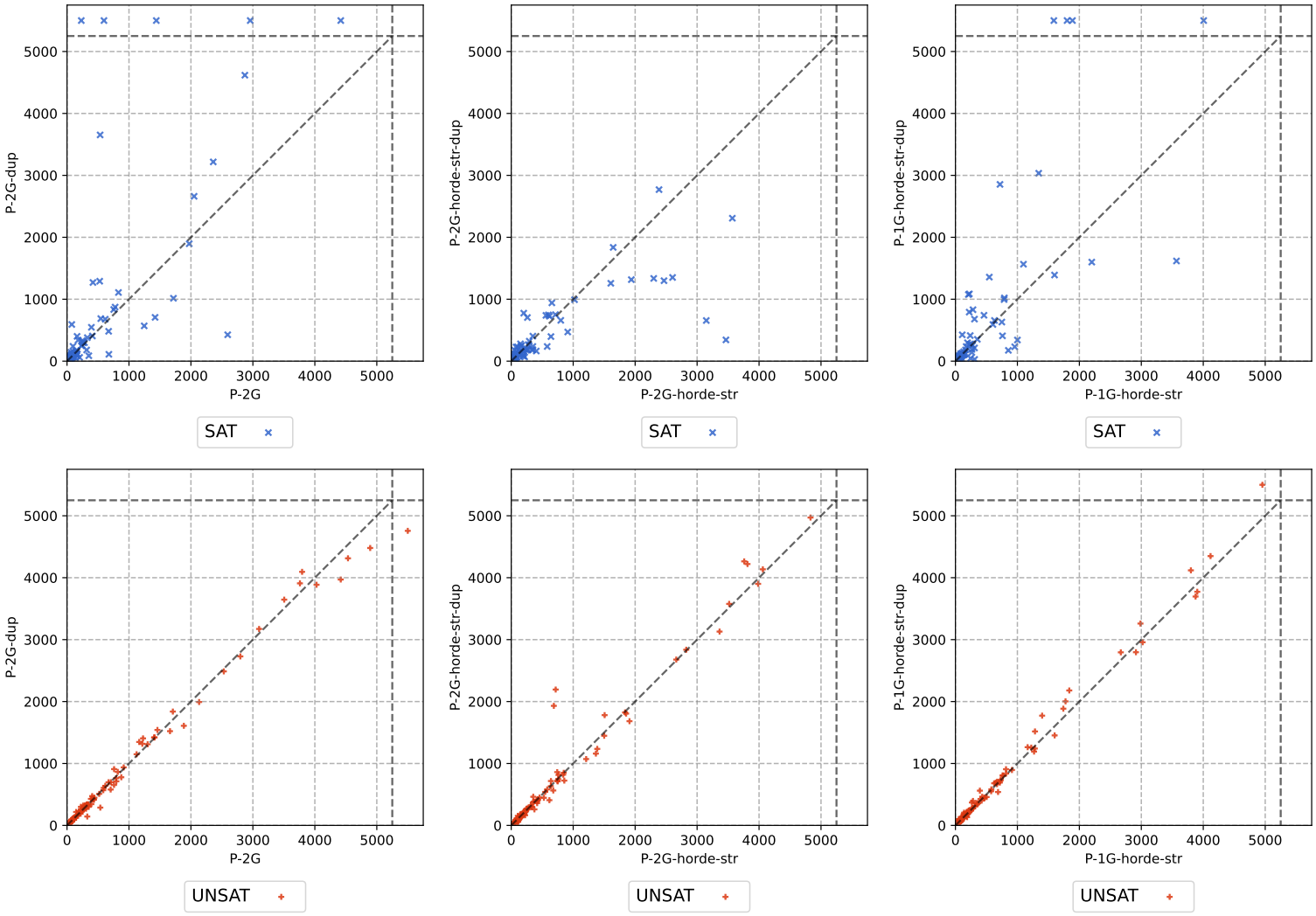


FIGURE 4.14 – Evaluation par instance de l'option `-dup`

souligner les performances remarquables de 2G-horde-str. Tout en maintenant sa supériorité pour la résolution des problèmes UNSAT, il se révèle être l'algorithme le plus efficace pour les problèmes SAT dans la configuration à 64 cœurs. Cela montre que l'impact négatif de l'option `-str` semble diminué par l'augmentation du nombre de cœurs disponibles.

4.4 CONCLUSION

Ce chapitre présente une implémentation de la minimisation de clauses [100] intégrée dans `Painless` [57]. Grâce à la modularité de `Painless`, nous avons pu tester l'efficacité de la minimisation dans différentes configurations de solveurs SAT parallèles. Dans cette étude, nous avons utilisé plusieurs stratégies de partage et différents paradigmes de parallélisation (*i.e.,* portfolio et divide-and-conquer). Nos expériences montrent qu'avoir un cœur dédié à la

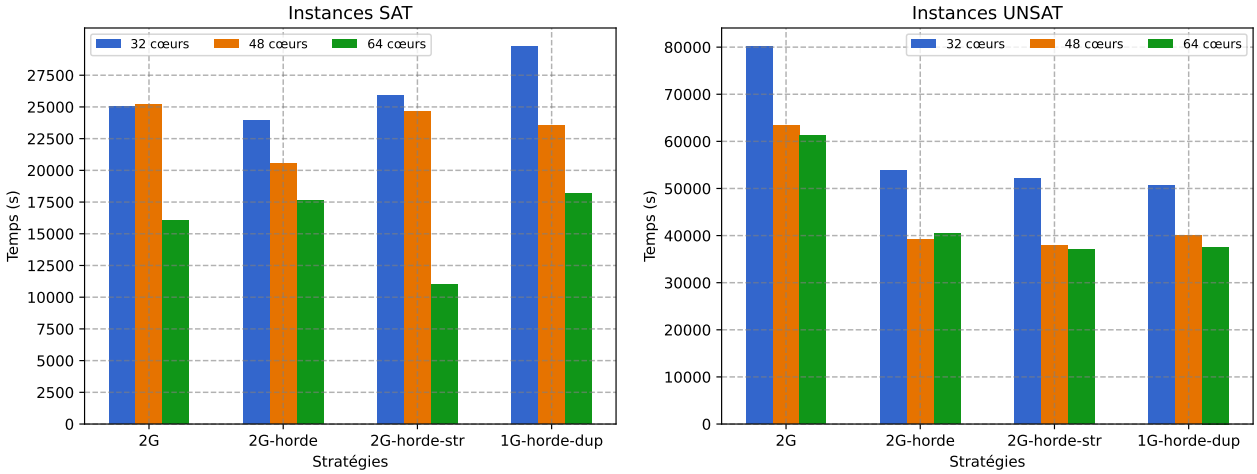


FIGURE 4.15 – Étude de passage à l'échelle

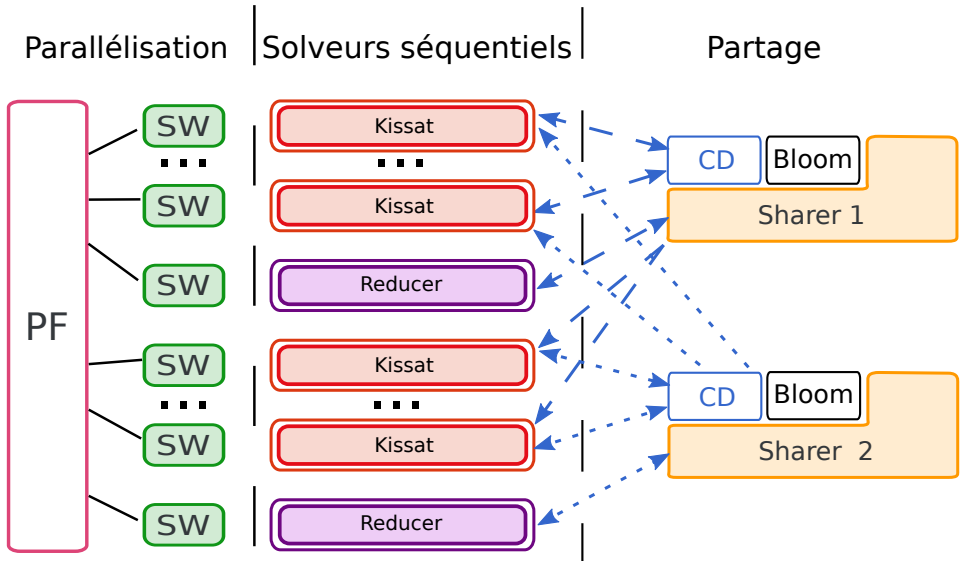


FIGURE 4.16 – Architecture finale après l'ajout de toutes les options (`P-2G-horde-str-dup`).

minimisation améliore les performances de nos solveurs parallèles, quelle que soit la configuration (y compris la configuration gagnante de la compétition SAT 2018).

En outre, nous avons combiné des techniques éprouvées et ajouté de nouveaux algorithmes pour améliorer la qualité du mécanisme de partage dans un solveur parallèle de pointe, à savoir `Parkissat-RS`. À l'implémentation originale, nous avons ajouté : un partitionnement du partage en introduisant des groupes de production, la stratégie de partage "horde", la minimisation et un mécanisme de suppression des duplications. Le solveur complet avec toutes les options est représenté à la Figure 4.16.

Nous avons expérimenté et comparé tous les modules complémentaires mis en œuvre sur les

benchmarks du concours SAT 2022. Cela nous a permis de tirer des conclusions sur l'effet global de chaque technique individuellement et après composition : pour les instances SAT, la meilleure implémentation est `P-2G`. Cela signifie globalement que l'implémentation originale est efficace et n'a pas besoin d'être modifiée. Seul le partitionnement des solveurs se révèle efficace, la division des solveurs au sein des groupes de production accélère significativement le partage. Pour les instances UNSAT, toutes les implémentations qui active l'option `-horde` sont très efficaces. Ce mécanisme se combine bien avec le renforcement et la suppression des doublons séparément, mais la combinaison des trois ne donne pas de résultats améliorés.

Toutes ces conclusions sont valables pour les machines multicœurs ($\leq$ 100 cœurs), et il sera très intéressant d'expérimenter ces stratégies dans le contexte des machines **many-core** ($>$ 100 cœurs). Dans de tels scénarios, le coût des mécanismes intensifs en CPU pourrait être compensé en utilisant un plus grand nombre de solveurs, permettant une meilleure utilisation des ressources.

Chapitre

5

Diversifier un solveur parallèle grâce au Bayesian Moment Matching

Contents

5.1 Introduction

Les algorithmes CDCL font encore aujourd'hui l'objet d'améliorations continues et sont adaptés à de nombreux domaines différents, tels que les mathématiques combinatoires et la cryptographie, qui étaient auparavant jugés trop difficiles. Bon nombre de ces améliorations sont des techniques d'optimisation par apprentissage machine (*ML pour machine learning*). Ce modèle de solveur SAT vu comme une collection de routines ML visant à séquencer, sélectionner et initialiser de manière optimale ses composants principaux (*e.g.,* branchement, redémarrage) a été articulé pour la première fois dans une série d'articles de Liang, Ganesh et al. [61, 60, 63], et développé par Ganesh et Vardi en 2020 [41]. Dans ce contexte, une variété de sous-routines de branchement, de redémarrage, d'initialisation et de séquençage ML ont été conçues et mises en œuvre dans de nombreux solveurs CDCL SAT au fil des ans [50].

Dans leur article [76], Duan et al. présentent un algorithme d'apprentissage probabiliste basé sur le Bayesian Moment Matching (BMM) et utilisé comme préprocesseur pour un solveur SAT de type CDCL. Le but de cet algorithme est de fournir une affectation initiale à partir de laquelle la recherche du solveur peut commencer. Le problème de trouver une affectation

initiale optimale pour démarrer la recherche d'un solveur est souvent appelé **problème d'initialisation**. La méthode BMM appliquée à la résolution du problème SAT, prend en entrée la formule à résoudre et une affectation de probabilité $P(x = T)$ pour chaque variable x qui indique la possibilité que cette variable soit vraie.

Dans ce contexte, la probabilité d'être vrai pour une variable x peut influer sur celle d'une autre variable y. De plus, la réalisation des variables se fait simultanément selon la loi de probabilité. Nous appelons la distribution de probabilités prise en entrée par la méthode BMM, la *distribution antérieure*. La méthode BMM applique l'inférence bayésienne à plusieurs reprises pour mettre à jour la distribution antérieure et utilise chaque clause de la formule d'entrée comme preuve afin de calculer une nouvelle distribution (*distribution postérieure*). La méthode produit une distribution représentant une affectation la plus à même de maximiser le nombre de clauses satisfaites (toutes, dans le meilleur des cas). Cette affection est ensuite utilisée pour initialiser les valeurs des variables et leurs ordres (de branchement). Bien que Dual et al. rapportent d'excellents résultats de l'utilisation de BMM comme préprocesseur pour résoudre le problème d'initialisation, ils ne l'ont pas utilisé d'une autre manière dans leur solveur.

Dans ce chapitre, nous proposons une technique basée sur BMM appliquée en cours de résolution pour mettre à jour les valeurs de phase et l'ordre des variables dans la routine de décision (en mettant à jour les heuristiques de branchement) d'un solveur CDCL. Notre méthode fonctionne comme suit : nous invoquons d'abord BMM en tant que préprocesseur qui prend en entrée une distribution aléatoire et la formule SAT à résoudre et renvoie une nouvelle distribution de probabilité (qui devient l'entrée pour les invocations ultérieures de la méthode BMM). Nous utilisons l'affectation suggérée par cette nouvelle distribution et mettons à jour la polarité des variables en conséquence. Par la suite, nous appelons BMM après la routine de propagation unitaire et avant de prendre une nouvelle décision (branchement) pour calculer et mettre à jour la distribution des probabilités et la polarité des variables ainsi que leurs ordres. Au cours de cet appel en cours de résolution, la méthode BMM dispose de clauses supplémentaires à traiter sous la forme de clauses apprises. L'intuition derrière notre méthode est que le raisonnement bayésien est un moyen puissant de guider la procédure de recherche CDCL loin des parties infructueuses de l'espace de recherche et vers les régions susceptibles de contenir des affectations satisfaisantes.

Comme indiqué précédemment, BMM utilise l'inférence bayésienne à partir d'une distribution de probabilité initialement aléatoire et les clauses de la formule comme preuve. Le choix de la distribution antérieure peut affecter la qualité de la distribution postérieure. Par conséquent, l'exécution du même solveur avec des graines initiales différentes pourrait conduire à des résultats de performance différents. Ce comportement peut être exploité dans le contexte d'un solveur parallèle de type Portfolio, où les solveurs sont exécutés avec des distributions aléatoires différentes. Il est bien connu que la diversification de l'heuristique de branchement en donnant un ordre de branchement ou des polarités initiales différentes, améliore considérablement les performances du solveur parallèle [13]. Par conséquent, dans ce chapitre, nous évaluons également l'utilisation de l'approche BMM pour la diversification des solveurs parallèles de type Portfolio.

Contributions.

1. Nous introduisons `slime-bmm`, un solveur séquentiel applicant l'algorithme CDCL augmenté d'une nouvelle routine BMM appliquée en temps que préprocesseur, mais aussi en cours de résolution.
2. Une étude expérimentale détaillée du solveur séquentiel `slime-bmm` sur un grand ensemble de problèmes cryptographiques par rapport au solveur `slime`, le gagnant du concours SAT 2021 sur la piste cryptographie. Nous montrons que pour certaines graines aléatoires, le solveur séquentiel `slime-bmm` surpasse largement le solveur séquentiel `slime` de base.
3. L'utilisation du solveur `slime-bmm` en tant que moteur séquentiel dans un solveur de type Portfolio parallèle, que nous appelons `p-slime-bmm`, implémenté dans le framework Painless [57]. Nous montrons que notre nouveau solveur hybride surpasse un Portfolio basé uniquement sur le solveur `slime`.

Structure du chapitre. Nous décrivons plus en détail la méthode BMM dans la Section 5.2. Section 5.3 présente comment la routine BMM est intégrée à l'algorithme CDCL. Section 5.4 évalue les performances de `slime-bmm` dans un contexte séquentiel sur des problèmes cryptographiques. Nous décrivons l'implémentation et les performances de `slime-bmm` au sein d'un solveur parallèle dans la Section 5.5. Finalement, la Section 5.6 conclut ce chapitre.

5.2 Bayesian Moment Matching

BMM est une technique utilisée pour approximer des distributions de probabilité complexes par des distributions plus simples. Elle est souvent utilisée lorsque la forme analytique exacte de la distribution a posteriori est difficile à obtenir ou coûteuse en termes de calcul. L'idée principale derrière BMM est de faire correspondre les moments (par exemple, la moyenne, la variance) de la distribution plus simple aux moments correspondants de la véritable distribution a posteriori. Ce faisant, nous cherchons à capturer les caractéristiques essentielles de la distribution a posteriori sans connaître explicitement sa forme complète. [76].

La formulation et l'application de BMM à SAT ont été proposées pour la première fois dans Duan et al. [35]. La méthode prend comme entrée une formule booléenne CNF et une distribution initiale de probabilité sur la polarité des variables de la formule d'entrée, et met à jour les probabilités de chaque variable en utilisant les clauses comme preuve. Plus précisément, chaque variable de la formule booléenne d'entrée ϕ se voit attribuer une variable aléatoire de Bernoulli, associée à une probabilité inconnue p qui représente la probabilité de la variable à être vraie ($P(x = T)$). Pour chaque clause de la formule, la croyance sur p est mise à jour en utilisant l'inférence bayésienne et le moment matching. Les clauses nous informent collectivement sur la façon dont la distribution cible devrait ressembler, de sorte que nous les utilisons comme preuve dans le processus d'inférence bayésien. Pour chaque clause de la formule, la croyance sur p est mise à jour en utilisant l'inférence bayésienne et *moment matching*. Les clauses nous informent collectivement sur la forme que devrait prendre la distribution cible, nous les utilisons donc comme preuves dans le processus d'inférence bayésienne.

Le processus BMM peut être résumé par les étapes suivantes :

1. Attribuer des probabilités à chaque variable : $P(x_i = T) \sim \mathtt{Beta}(\alpha_i, \beta_i)$
2. Choisir une clause C_j et calculer la probabilité d'évaluer la clause comme vraie (de satisfaire C_j) étant donné les probabilités des variables : $P(C_j = T|x_1, \cdots, x_n)$
3. Appliquer l'inférence bayésienne pour mettre à jour les probabilités des variables de C_j en se basant sur la probabilité de satisfaire C_j : $P(x_i = T|C_j = T)$. À ce stade, nous obtenons un modèle de mélange avec deux composantes Beta pour chaque variable. Approximer la distribution apprise avec une seule distribution modale pour éviter une explosion exponentielle.
4. Utiliser *moment matching* avec les premier et second moments (moyenne et variance respectivement) pour projeter sur les distributions de $\mathtt{Beta}$ de l'étape 1.
5. Répéter les étapes 1-4 K fois. K est un hyperparamètre de l'algorithme, plus sa valeur est grande, plus les probabilités sont précises, mais plus le coût de calcul est élevé.

Après que BMM ait scanné toutes les clauses d'entrées, il arrive à une distribution postérieure qui suggère une affectation qui peut idéalement satisfaire la plupart des clauses, si ce n'est toutes.

On peut traiter une telle affectation comme une bonne estimation d'une affectation satisfaisant la formule ϕ (en assumant qu'elle est satisfaisable). Il a été démontré qu'utiliser BMM comme algorithme de préprocessing pour initialiser la polarité et l'ordre des variables améliore les performances de différents solveurs [35], en particulier sur les instances satisfiables venant de problèmes issues du domaine de la cryptographie. Dans ce chapitre, nous allons plus loin et utilisons la méthode BMM à des points particuliers de la résolution pour mettre à jour les polarités et l'ordre des variables.

5.3 DESCRIPTION DE L'ALGORITHME

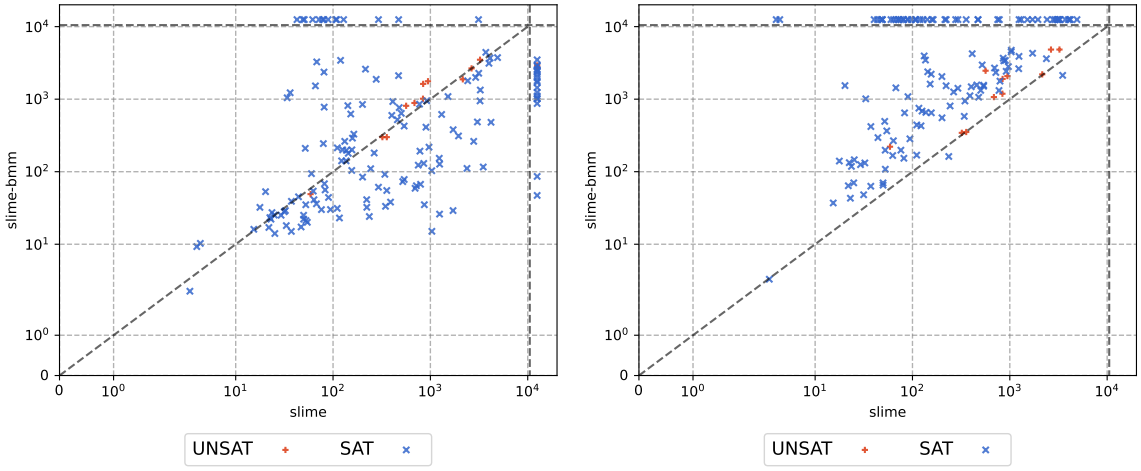
Algorithme 5 reprend l'algorithme CDCL déjà expliqué dans ce manuscrit (Algorithme 3). Comme expliqué dans l'introduction, nous augmentons l'algorithme susmentionné avec la procédure BMM utilisée comme préprocesseur, ainsi qu'en tant que routine permettant de mettre à jour la phase et l'ordre de branchement des variables. L'idée de notre contribution est de guider le solveur à des points clés de sa progression. Les instructions grisées dans cet algorithme sont les instructions rajoutées pour intégrer la technique BMM dans l'algorithme CDCL.

Préprocesseur : BMM est appelé pour initialiser la polarité et l'ordre de branchement des variables (Line 4). Durant ce prétraitement, les entrées de la procédure BMM sont les clauses de la formule et une distribution antérieure générée de manière aléatoire. Parce que cette étape est exécutée une fois, nous pouvons nous permettre un coût de calcul plus élevé afin d'obtenir une distribution postérieure plus précise. Ainsi, le nombre de passes sur l'ensemble des clauses est défini par $K = 100$.

```
1  function CDCL(φ : formule CNF)
       /* returns ⊤ si φ est SAT sinon ⊥ (UNSAT)                                  */
2      α ← ∅                                                  // Affectation courante
3      lvl ← 0                                                // Niveau de décision courant
4      miseAjourBMM()                                         // Appeler BMM avec K = 100
5      Indéfiniment
6          (φ′, α′) ← propagationUnitaire(φ|α)
7          α ← α ∪ α′                  // Ajouter les littéraux propagés à l'affectation α
8          si φ′ = ∅ alors
9              retourner true                                                  // φ est SAT
10         si ∅ ∈ φ′ alors                                    // Il y a un conflit à analyser
11             si lvl = 0 alors
12                 retourner false                                             // φ est UNSAT
13             C ← analyseDeConflit(φ, α)
14             φ ← φ ∪ {C}
15             lvl ← backjumpEtRedemarrage(lvl, C, ...)
16             α ← {ℓ ∈ α | δ(ℓ) ≤ lvl}
17         sinon
18             si le seuil limite est atteint alors
19                 miseAjourBMM()                                  // Appeler BMM avec K = 10
20             α ← α ∪ {choixLitteralBranchement()}   // Choix d'un nouveau
                   littéral de branchement
21             lvl ← lvl + 1
```

Algorithme 5 : L'algorithme CDCL augmenté du Bayesian Moment Matching. L'appel à BMM en cours de résolution se fait après la propagation unitaire, indiqué en caractères gris ci-dessus.

En cours de traitement : Lorsque la propagation unitaire atteint un point fixe (Ligne 19), c.-à-d. il n'y a plus de clauses unitaires à propager et il n'y a pas de conflit, la procédure BMM est appelée pour réévaluer les probabilités pour toutes les variables. La différence ici est que la distribution antérieure en entrée prend en compte l'affectation courante des variables (la probabilité des variables qui ont une valeur dans l'affectation partielle courante est fixée à 0 ou 1 selon cette valeur). Le reste des variables sont mises à jour selon cette nouvelle distribution antérieure. Une autre différence est que la procédure ne prend pas juste les clauses du problème en entrée, elle peut aussi profiter des informations apprises par le solveur entre chaque appel. Pour maximiser la qualité de l'information sans rendre le volume des clauses à explorer par la procédure trop élevé, nous utilisons uniquement les clauses apprises dont la valeur LBD est inférieure ou égale à 3. Contrairement à l'appel en prétraitement, cette étape peut être exécutée plusieurs fois au cours de l'exploration de l'espace de recherche par le solveur. Pour limiter le coût en calcul, la limite sur le nombre de passes à effectuer sur les clauses en entrée est définie



(a) Best runs **(b)** Worst runs

FIGURE 5.1 – Diagramme de dispersion montrant les performances des solveurs séquentiels **`slime-bmm`** vs `slime` sur la piste "Cryptographie" de la compétition SAT 2021.

par $K = 10$.

La limite imposée sur le nombre d'itérations de la procédure n'est pas suffisante pour contenir la pénalité considérable en coût de calcul qu'elle engendre si elle est appelée trop souvent. Par conséquent, nous appelons la procédure lorsqu'un certain seuil de progression est atteint. Ce seuil dépend du nombre de redémarrages d'une part afin de ne pas appeler la procédure trop souvent et du nombre de variables dans l'affectation partielle courante afin d'appeler la procédure à un point important de l'exploration de l'espace de recherche. Nous considérons ce seuil atteint lorsque le solveur a redémarré 50 fois et que le nombre de variables dans l'affectation courante excède 40% du nombre total de variables ou 90% de l'affectation la plus grande vu jusqu'à présent. Ces *"nombres magiques"* reprenant les résultats du solveur depuis lequel nous avons basé notre approche. À la fin de la procédure, la distribution postérieure apprise pour toutes les variables est utilisée pour aider le solveur à développer davantage l'arbre de recherche.

L'étape de préprocessing est similaire au préprocesseur présenté dans [35], cependant, l'appel de BMM en cours de résolution est conçu différemment. Dans le papier original décrivant l'approche BMM, les auteurs mettent uniquement à jour les probabilités lorsqu'une clause unitaire ou binaire est apprise, utilisant ces clauses comme de nouvelles preuves. En effet, les clauses unitaires et binaires sont des informations précieuses apprises par le solveur sur le sous-espace de recherche qu'il explore. Nous pensons qu'il est tout aussi important d'aider à guider la recherche du solveur CDCL lorsque l'il atteint un point où les perspectives d'apprentissage sont moindres. Par conséquent, nous avons implémenté cette mise à jour par la méthode BMM pour guider la recherche du solveur à un point particulier de l'algorithme, lorsque la propagation unitaire n'engendre pas d'apprentissage.

5.4 ÉVALUATION DES SOLVEURS SÉQUENTIELS SLIME ET SLIME-BMM

Nous avons choisi de comparer l'efficacité de notre version de la méthode BMM sur des problèmes provenant du domaine de la cryptographie, compte tenu de son succès antérieur dans ce domaine. Pour comparaison, nous avons choisi le vainqueur de la compétition SAT 2021 dans la catégorie cryptographie [1], appelé slime [84]. L'implémentation de l'algorithme CDCL de ce solveur a servi de base pour notre implémentation.

Il est important de noter que slime utilise une méthode de réévaluation de la polarité et l'ordre des variables similaire à celle présentée dans ce chapitre, cependant leur méthode se base sur l'utilisation d'un algorithme de résolution SAT appelé *Stochastic Local Search (SLS)* [51, 26]. Contrairement aux algorithmes dérivés de la procédure de Davis-Putnam-Logemann-Loveland (DPLL) qui progressent vers une solution en affectant une variable à la fois, les algorithmes SLS naviguent d'une affectation complète de la formule à une autre en modifiant la valeur d'une variable, généralement dans le but de minimiser le nombre de clauses actuellement insatisfaites. La plupart des implémentations de cet algorithme sont incomplètes, car elles ne sont pas capables de trouver l'insatisfiabilité de la formule et ne fournissent aucune garantie de trouver une solution dans le cas SAT, bien qu'il soit possible de rendre l'algorithme complet [39]. Dans slime, SLS est également utilisé lorsque la propagation unitaire atteint un état particulier afin de guider l'exploration de l'espace de recherche. De ce fait, nous avons remplacé ce composant par notre procédure BMM ainsi que la suppression de toutes les heuristiques liées à l'utilisation du solveur SLS. Nous appelons cette version du solveur slime-bmm.

La configuration de slime qui a gagné la compétition est déterministe [84], nous l'avons donc lancé sur ces instances une seule fois. Inversement, l'initialisation du composant BMM induit un caractère aléatoire comme décrit dans Section 5.2, nous avons lancé slime-bmm 10 fois pour chaque instance.

Le diagramme de dispersion de la Figure 5.1 montre les résultats de notre expérience. Figure 5.1a (respectivement Figure 5.1b) met en évidence les scores des solveurs par rapport aux meilleures (respectivement pires) exécutions de slime-bmm. Les axes de ces figures représentent le temps d'exécution en secondes sur une échelle logarithme. Nous ne représentons pas sur la figure les cas où les deux solveurs ne parviennent pas à résoudre une instance dans le temps imparti. Nous pouvons donc nous concentrer sur les instances où l'un des deux solveurs se démarque. Bien que la limite sur la durée d'exécution soit fixée à 5000 secondes, pour plus de lisibilité, une exécution qui n'aboutit pas à une solution est représentée par un point au-delà de la ligne pointillée, au-dessus de 10^4 secondes.

Nous pouvons observer que le cas des meilleures exécutions, slime-bmm est plus efficace que slime, en résolvant 6 instances supplémentaires et étant de manière générale plus rapide sur de nombreuses instances. Cependant, lorsque l'on s'intéresse aux pires cas, slime est plus stable et surpasse les performances de slime-bmm.

1. https ://satcompetition.github.io/2021/

Nous tirons deux conclusions de ces résultats expérimentaux :

1. L'utilisation de BMM permet de guider un solveur CDCL en cours de résolution. Comparé à la méthode basée sur la SLS utilisée par `slime`, BMM est efficace sans nécessiter autant d'ajustement d'heuristiques.

2. L'exécution de solveurs séquentiels en utilisant des distributions antérieures aléatoires différentes pour la routine BMM leur permet d'explorer différentes solutions (pour des instances satisfiables), ce qui constitue une grande opportunité pour la création d'une méthode de diversification pour un solveur parallèle de type Portfolio. C'est cette idée que nous explorons dans la section suivante, où nous créons et évaluons un solveur parallèle de type Portfolio dont l'exploration des travailleurs est diversifiée en utilisant une graine différente pour l'initialisation de la distribution initiale de BMM.

5.5 ARCHITECTURE DU SOLVEUR PARALLÈLE `p-slime-bmm` ET SES RÉSULTATS

Dans le but d'implémenter le solveur parallèle Portfolio que nous appelons `p-slime-bmm`, nous utilisons encore une fois le framework Painless (Section 2.3.4). Il était simple d'intégrer nos deux solveurs séquentiels `slime` et `slime-bmm` en temps que travailleurs au sein d'un solveur parallèle. Les solveurs parallèles résultants sont appelés `p-slime-bmm` et `p-slime`.

Les solveurs CDCL utilisés en tant que travailleurs dans le solveur parallèle `p-slime-bmm` sont soit des `slime` ou des `slime-bmm`. Nous faisons varier la proportion de `slime-bmm` entre les différentes versions du solveur de 25% à 90%. Le but est d'assurer une collaboration totale entre les solveurs du Portfolio pour résoudre le maximum d'instances. Quant à `p-slime`, un solveur séquentiel déterministe ne ferait pas sens dans le contexte parallèle, nous utilisons donc le mécanisme de diversification de `slime`, qui consiste à fixer une polarité aléatoire pour chaque variable, pour chaque `slime` du Portfolio.

Ces deux solveurs sont lancés sur un cluster de 12-core Intel Xeon CPU E5645, avec 64 Go de RAM et un délai de 5000s. Dans cette étude de performance, nous utilisons les mesures de succès suivantes : le *Penalized Average Runtime*(PAR-2) qui additionne le temps d'exécution

Solveurs	PAR2	UNSAT	SAT	TOTAL (200)
p-slime-bmm-50%	133H36	13	147	160
p-slime-bmm-75%	136H05	13	145	158
p-slime-bmm-90%	138H39	13	144	157
p-slime	142H03	13	142	155
p-slime-bmm-25%	147H42	13	140	153

TABLE 5.2 – Cette table montre les performances de `p-slime` et `p-slime-bmm` sur la piste "Crypto" de la compétition SAT 2021.

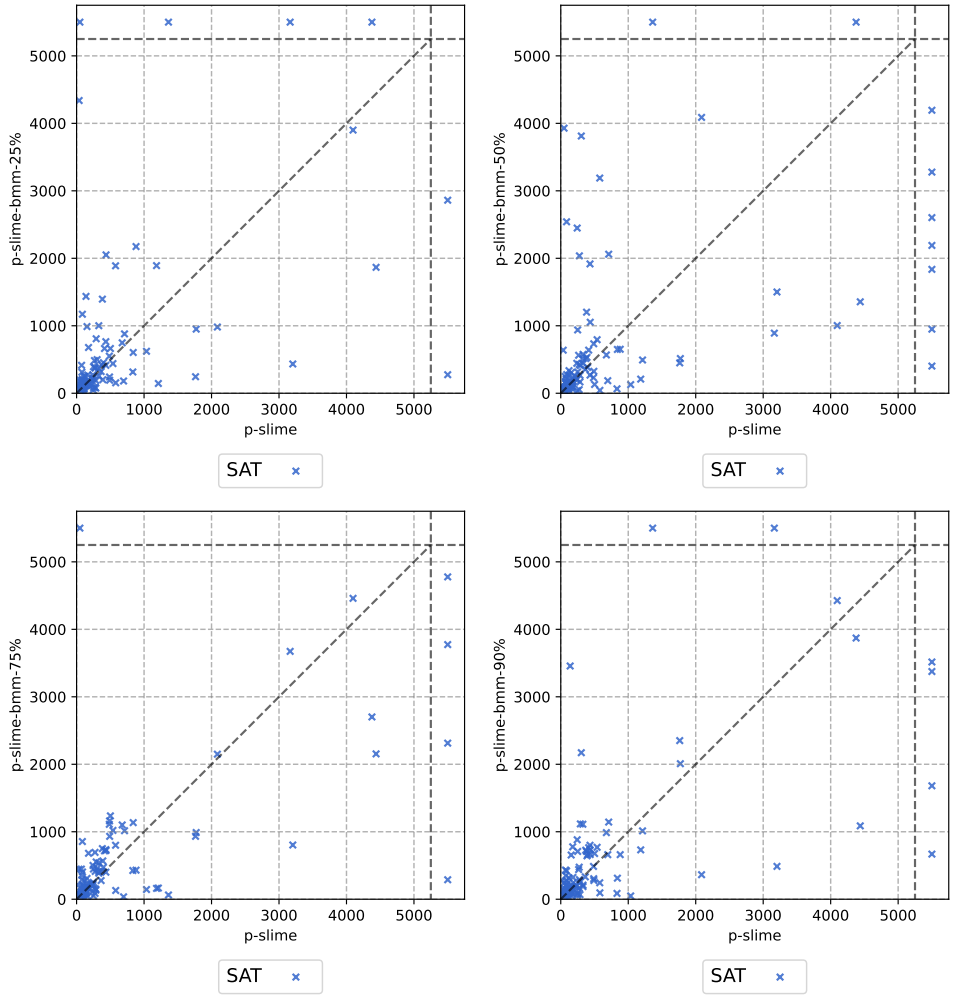


FIGURE 5.3 – Evaluation par instance SAT de `p-slime-bmm`

d'un solveur et pénalise les exécutions qui dépassent le délai avec un facteur 2 ; le nombre d'instances résolues.

Comme on peut l'observer dans la Table 5.2, il semble que le fait d'avoir les deux algorithmes en proportion égale constitue un bon équilibre. En effet, `p-slime-bmm-50%` résout 5 instances SAT supplémentaires par rapport à `p-slime`. Par conséquent, le nouveau solveur proposé s'avère plus efficace que la version parallèle du solveur servant de base `p-slime`. Les Figures 5.3 et 5.4 montrent qu'une différence entre les solveurs est présente sur les instances SAT et non les instances UNSAT. Cela peut s'expliquer par le fait que BMM dirige naturellement vers une solution et est donc moins utile s'il n'y en a pas. Il est rassurant de voir que BMM n'affecte pas négativement la résolution des instances UNSAT.

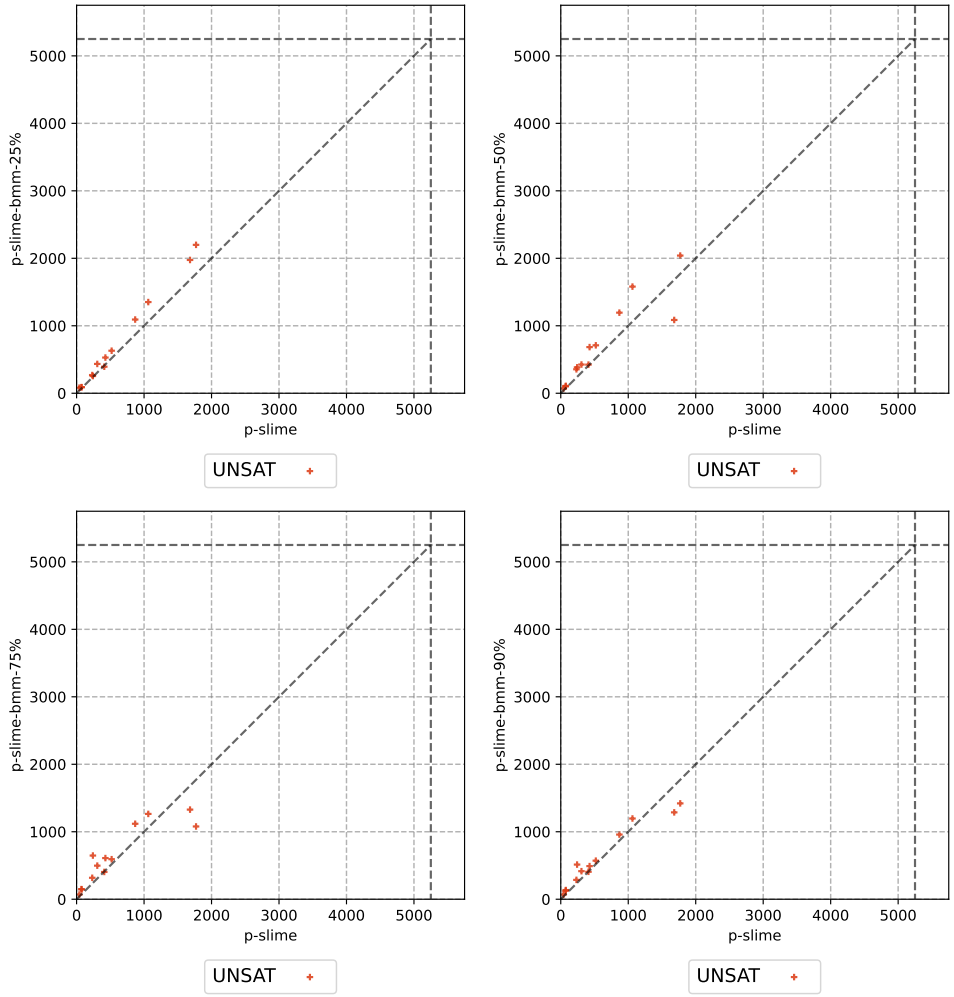


FIGURE 5.4 – Evaluation par instance UNSAT de `p-slime-bmm`

5.6 CONCLUSION

Dans ce chapitre, nous avons présenté une technique de traitement en cours de résolution basée sur le Bayesian Moment Matching (BMM) pour des solveurs CDCL. Nous relançons ce traitement BMM avant le branchement suivant une propagation unitaire sans conflit selon certaines conditions, dans le but de mettre à jour la polarité et l'ordre de branchement des variables. L'intuition clé qui sous-tend notre méthode est que le raisonnement bayésien est un moyen puissant de guider la procédure de recherche CDCL loin des parties infructueuses de l'espace de recherche d'une formule propositionnelle satisfaisable, et vers les régions qui sont susceptibles de contenir des affectations qui forment un modèle pour la formule.

Nous avons massivement expérimenté notre approche sur des instances cryptographiques et dans un cadre séquentiel. Les résultats sont positifs pour certaines graines aléatoires et pas pour d'autres (étant donné la nature probabiliste de BMM). Cela nous a conduit à développer

un solveur parallèle de type Portfolio en faisant varier le nombre de travailleurs utilisant la méthode BMM. Les solveurs parallèles hybrides qui en résultent montrent de bonnes performances par rapport au solveur original.

En utilisant une approche incrémentale, nous avons trouvé que 50% de `slime-bmm` est une bonne proportion, mais ce n'est peut-être pas la solution la plus optimale. De futurs travaux pourraient affiner la proportion de `slime-bmm` dans le Portfolio en se basant sur le travail effectué dans [64] en formulant le problème sous la forme du problème du bandit manchot.

Une autre piste d'amélioration est d'affiner l'heuristique décidant quand relancer la procédure BMM (actuellement, l'heuristique utilisée est empruntée du solveur `slime`). Nous pensons qu'une étude approfondie du comportement du solveur nous permettra de définir des seuils plus appropriés à notre méthode. Nous pensons, en particulier pouvoir exploiter le concept de *phase de dépression*, initié dans [29], pour concevoir un seuil d'activation pourrait être une direction très fructueuse. Une phase de dépression est une longue séquence de branchement qui ne mène à aucun conflit. L'idée serait d'appeler la mise à jour du BMM chaque fois que les phases de dépression ont atteint une certaine limite prédéfinie.

Chapitre

6

Conclusion et perspectives

6.1 Contributions

Cette thèse s'inscrit dans la continuité des nombreux efforts déployés pour améliorer la capacité de résolution des solveurs SAT parallèles. Un solveur SAT parallèle peut être optimisé en fonction de plusieurs composants :

1. L'algorithme séquentiel effectivement mis en œuvre sur plusieurs cœurs de calcul.
2. La répartition des tâches entre les différents travailleurs.
3. Le partage d'informations entre les travailleurs.

Tous ces aspects peuvent être examinés selon deux axes : l'axe "logique" et l'axe système. Ces deux axes représentent la distinction entre l'amélioration d'un mécanisme par le biais d'algorithmes ou d'heuristiques sophistiquées déduits de définitions formelles et d'études expérimentales approfondies d'une part, et son amélioration par l'utilisation efficace de structures de données en mémoire ou une gestion appropriée du parallélisme d'autre part. Au cours de cette thèse, nous nous sommes appuyés sur la plateforme `Painless` pour la gestion du parallélisme et l'évaluation rigoureuse de nos travaux. Ainsi nous avons pu nous consacrer à l'étude de mécanismes visant à améliorer le partage des clauses au sein de deux contributions, puis à l'intégration d'un composant permettant d'accroître la capacité de résolution d'un solveur dans une dernière contribution.

Notre première contribution a amélioré le partage d'informations en identifiant finement les clauses partagées de *haute qualité*. Ces clauses, lorsqu'elles sont partagées entre les différents solveurs d'une stratégie de résolution parallèle, devraient conduire à une amélioration des performances. Certaines métriques de performance ont émergé, comme la LBD, mais elles dépendent fortement de l'état local du solveur d'origine. La valeur de ces métriques une fois que la clause est importée par un autre solveur séquentiel peut être totalement incohérente par rapport à sa progression. Nous avons proposé une nouvelle métrique visant à identifier les clauses de haute qualité apprises, ainsi qu'une politique de partage des clauses basée sur la combinaison d'une métrique locale, la LBD, avec la structure des communautés mises en

évidence par les formules SAT. Nous avons développé une méthodologie permettant d'analyser a priori les caractéristiques des clauses apprises en séquentiel et avons extrait un filtre en parallèle. Nous avons considéré après avoir dûment examiné nos résultats expérimentaux, que partager toutes les clauses avec une LBD <= 3 ainsi que les clauses ayant à la fois LBD = 4 et COM <= 3 est une bonne stratégie de partage. Cette stratégie montre des résultats corrects sur un large benchmark.

Notre deuxième contribution a amélioré le partage d'informations, cette fois en réduisant la taille de l'information partagée. En effet, il est bien connu que plus les clauses apprises sont courtes, plus leur capacité à réduire l'espace de recherche est élevée. Cette contribution se décompose en deux parties. La première partie a intégré un composant permettant la minimisation asynchrone des clauses apprises au sein d'un solveur parallèle. L'évaluation de ce composant montre d'excellents résultats dans différentes configurations de solveurs SAT parallèles sur le benchmark de la compétition 2018. Dans une deuxième partie, nous avons évalué l'intégration de différents mécanismes de partage, issus de l'état de l'art, ainsi que de la minimisation, dans un solveur parallèle plus récent que celui utilisé dans la première partie de l'étude. Ces mécanismes comprennent un filtre dynamique pour le partage des clauses et la suppression des doublons. Les résultats de cette étude ont démontré que la minimisation asynchrone, bien qu'elle soit moins efficace que dans la première expérience, présente néanmoins un intérêt pour le traitement des instances UNSAT et se montre particulièrement performante lorsque le nombre de ressources CPU disponibles est augmenté. De plus, cette étude a révélé une stagnation des performances pour les instances UNSAT lorsque le nombre de cœurs de calcul augmente.

Notre dernière contribution a amélioré l'algorithme séquentiel sous-jacent d'un solveur parallèle. Nous présentons et évaluons une technique basée sur le Bayesian Moment Matching appliquée au cours de la résolution pour paramétrer la routine de décision (routine qui détermine quelle sera la prochaine variable à affecter et à quelle valeur) d'un solveur. En particulier, le comportement initialement aléatoire de ce mécanisme est exploité dans le développement d'un solveur parallèle (en utilisant une graine différente pour initialiser la distribution de probabilité pour chaque solveur). Ce nouveau solveur parallèle s'est montré performant sur des instances de problèmes cryptographiques de la compétition SAT 2021.

6.2 PERSPECTIVES

Dans cette section, nous présentons quatre perspectives de recherche visant à prolonger ou compléter nos travaux.

Perspective 1 : Nous estimons qu'il est nécessaire de compléter notre travail sur les communautés en adoptant une approche plus dynamique dans notre méthode de filtrage. En effet, notre deuxième contribution a démontré l'importance d'un filtre dynamique grâce à la stratégie `HordeSat`. Il serait envisageable d'utiliser un delta autour d'une valeur seuil spécifique pour

la LBD, tout en maintenant notre seuil pour COM. Une telle stratégie pourrait être évaluée avec les autres options présentées dans le Chapitre 4.

Perspective 2 : Notre analyse des résultats Chapitre 4 a révélé la difficulté de faire passer à l'échelle notre configuration de `Painless`. Nos conclusions sont valables pour les machines multicœurs ($\leq$ 100 cœurs), et il serait pertinent d'expérimenter de nouvelles stratégies dans le contexte des machines **many-core** ($>$ 100 cœurs). Nous devrons d'abord identifier les raisons logiques ou matérielles qui causent ce blocage et les corriger, notamment en repensant le partage. Nous envisageons de repenser totalement la façon dont l'algorithme séquentiel sous-jacent d'un solveur parallèle stocke les clauses apprises, afin de créer une base de données globale de clauses partagée entre chaque travailleur.

Perspective 3 : Nous envisageons d'exploiter une ressource matérielle ignorée au cours de cette thèse, à savoir le GPU. Nous pensons l'utiliser pour effectuer des calculs massifs impossibles à réaliser sans ralentir la résolution dans un solveur n'utilisant que le CPU. L'idée serait d'augmenter le partage des clauses à l'aide d'un oracle capable de décider si une clause est localement intéressante pour un potentiel consommateur et peut donc être ajoutée à sa base de données. L'algorithme de cet oracle est capable d'extraire des caractéristiques simples de la clause dans le contexte de l'affectation courante du travailleur consommateur considéré, par exemple si la clause est rendue *unitaire* par l'affectation courante ou si elle est déjà en conflit. Si ce calcul est simple, il doit être effectué sur des dizaines de millions de clauses pour chaque consommateur potentiel, ce qui nécessite une énorme capacité de calcul. Il est alors intéressant d'utiliser les GPU, qui offrent une capacité de calcul massivement parallèle et ont déjà été exploités dans le passé pour résoudre le problème SAT [34, 31].

Perspective 4 : Nous souhaitons totalement repenser `Painless` afin de le rendre compatible avec une exécution dans un environnement distribué. Dans ce contexte, le solveur devra tourner sur une grappe de machines, potentiellement hétérogènes et géographiquement dispersées. Afin de tirer parti de telles machines, nous devons mettre à jour `Painless` qui, dans son implémentation courante, utilise la mémoire partagée entre les threads. Nous devons le transformer en un framework capable de combiner la communication locale, plus rapide grâce à la mémoire partagée, avec la communication à distance utilisant MPI. Une approche basique d'une telle architecture a été présentée lors de la compétition de solveurs SAT distribués 2021 [98], mais n'a pas atteint le podium, malgré le succès de sa version parallèle.

De manière générale, il faut repenser les mécanismes de division du travail et de partage utilisés dans un contexte distribué. Des travaux antérieurs ont déjà abordé cette tâche avec succès, notamment le solveur Mallob [85], qui s'est distingué lors de toutes les compétitions SAT distribuées à ce jour (2020 à 2022 [1]). Ce solveur est capable d'équilibrer dynamiquement la charge de travail entre les différentes machines et de résoudre plusieurs instances simultanément. De plus, il présente une méthode très efficace de distribution des clauses apprises. Celles-ci sont transmises et agrégées le long d'un arbre binaire dont les nœuds représentent

une machine. Lors de la remontée des nœuds vers la racine pour créer un ensemble unique de clauses à partager, une étape de réduction est effectuée à chaque nœud afin de privilégier les clauses de petite taille et de supprimer les doublons à l'aide d'un filtre de Bloom. Ainsi, une fois de plus, la manière de paramétrer le partage est cruciale, et nous devrons réfléchir à la manière d'intégrer les différents mécanismes présentés dans le Chapitre 4 ainsi que ceux présentés dans nos perspectives.

Table des figures

Bibliographie

[1] Sat competitions. https://satcompetition.github.io/.

[2] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 410–423. Springer, 2012.

[3] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 238–254. Springer, 2015.

[4] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette. Dolius : A distributed parallel sat solving framework. In *Pragmatics of SAT International Workshop (POS) at SAT*, pages 1–11. Citeseer, 2014.

[5] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 188–200. Springer, 2011.

[6] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel sat solver. In *Proceedings of the 22th International Conference of Principles and Practice of Constraint Programming (CP)*, pages 30–48. Springer, 2016.

[7] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conferences on Artifical Intelligence (IJCAI)*, pages 399–404. AAAI Press, 2009.

[8] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *Proceedings of the 18th International Conference of Principles and Practice of Constraint Programming (CP)*, pages 118–126. Springer, 2012.

[9] Gilles Audemard and Laurent Simon. Glucose in the sat 2014 competition. In *Proceedings of SAT Competition 2014 : Solver and Benchmark Descriptions*, page 31. Department of Computer Science, University of Helsinki, Finland, 2014.

[10] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205. Springer, 2014.

[11] Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat : A massively parallel portfolio sat solver. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 156–172. Springer, 2015.

[12] Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.

[13] Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.

[14] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 28–33. Springer, 2008.

[15] Armin Biere. Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. In *Proceedings of SAT Competition 2016 : Solver and Benchmark Descriptions*, page 44. Department of Computer Science, University of Helsinki, Finland, 2016.

[16] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. In *Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions*, pages 13–14. Department of Computer Science, University of Helsinki, Finland, 2018.

[17] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207. Springer, 1999.

[18] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. In *Proceedings of sat competition 2020 : Solver and benchmark descriptions*. University of Helsinki, Department of Computer Science, 2020.

[19] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

[20] Armin Biere and Carsten Sinz. Decomposing sat problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4) :201–208, 2006.

[21] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics : Theory and Experiment*, 2008(10) :P10008, 2008.

[22] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[23] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[24] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2) :172–188, 2007.

[25] Michael Buro and H Kleine Büning. Report on a sat competition. Technical Report 110, Department of Mathematics and Informatics, University of Paderborn, Germany, 1992.

[26] Shaowei Cai, Chuan Luo, and Kaile Su. Ccanr : A configuration checking based local search solver for non-random satisfiability. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 1–8, Cham, 2015. Springer International Publishing.

[27] Shaowei Cai and Xindi Zhang. Four relaxed cdcl solvers. In *Proceedings of SAT RACE 2019 : Solver and Benchmark Descriptions*, pages 35–36. Department of Computer Science, University of Helsinki, Finland, 2019.

[28] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining VSIDS and CHB Using Restarts in SAT. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20 :1–20 :19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[29] Solimul Chowdhury, Martin Müller, and Jia You. Guiding cdcl sat search via random exploration amid conflict depression. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 :1428–1435, 04 2020.

[30] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[31] Alessandro Dal Palu, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Cud@ sat : Sat solving on gpus. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3) :293–316, 2015.

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

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

[34] Hervé Deleau, Christophe Jaillet, and Michaël Krajecki. Gpu4sat : solving the sat problem on gpu. In *the 9th International Workshop on State of the Art in Scientific and Parallel Computing (PARA)*, 2008.

[35] Haonan Duan, Saeed Nejati, George Trimponias, Pascal Poupart, and Vijay Ganesh. Online bayesian moment matching based sat solver heuristics. In *International Conference on Machine Learning*, pages 2710–2719. PMLR, 2020.

[36] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. Sat versus unsat. In *Second DIMACS Implementation Challenge*, volume 26, pages 415–436. 1996.

[37] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518. Springer, 2003.

[38] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 61–75. Springer, 2005.

[39] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. In *AAAI Conference on Artificial Intelligence*, 2004.

[40] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, United States of America, 1995.

[41] Vijay Ganesh and Moshe Y. Vardi. On the unreasonable effectiveness of SAT solvers. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 547–566. Cambridge University Press, 2020.

[42] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel sat solving. In *Proceedings of the 16th International Conference of Principles and Practice of Constraint Programming (CP)*, pages 252–265. Springer, 2010.

[43] Long Guo and Jean-Marie Lagniez. Dynamic polarity adjustment in a parallel sat solver. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 67–73. IEEE, 2011.

[44] Youssef Hamadi, Said Jabbour, and Jabbour Sais. Control-based clause sharing in parallel sat solving. In *Autonomous Search*, pages 245–267. Springer, 2011.

[45] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat : a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4) :245–262, 2009.

[46] Hyojung Han and Fabio Somenzi. Alembic : An efficient algorithm for cnf preprocessing. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 582–587, New York, NY, USA, 2007. Association for Computing Machinery.

[47] Marijn Heule and Hans van Maaren. Look-ahead based sat solvers. *Handbook of satisfiability*, 185 :155–184, 2009.

[48] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 201–215, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer : Guiding cdcl sat solvers by lookaheads. In *Proceedings of the 11th Haifa Verification Conference (HVC)*, pages 50–65. Springer, 2011.

[50] Sean B. Holden. Machine learning for automated theorem proving : Learning to solve SAT and QSAT. *Found. Trends Mach. Learn.*, 14(6) :807–989, 2021.

[51] Holger H Hoos and Thomas Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for sat. *Artificial Intelligence*, 112(1) :213–232, 1999.

[52] Antti EJ Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving sat in grids. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 430–435. Springer, 2006.

[53] Robert G Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4) :167–187, 1990.

[54] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, volume 92, pages 359–363, 1992.

[55] Davide Lanti and Norbert Manthey. Sharing information in parallel search with search space partitioning. In *Proceedings of the 7th International Conference on Learning and Intelligent OptimizatioN (LION)*, pages 52–58. Springer, 2013.

[56] Ludovic Le Frioux. *Towards more efficient parallel SAT solving*. Theses, Sorbonne Université, July 2019.

[57] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Painless : a framework for parallel sat solving. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 233–250. Springer, 2017.

[58] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Modular and efficient divide-and-conquer SAT solver on top of the Painless framework. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 135–151. Springer, 2019.

[59] Ludovic Le Frioux, Hakan Metin, Souheib Baarir, Maximilien Colange, Julien Sopena, and Fabrice Kordon. painless-mcomsps and painless-mcomsps-sym. In *Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions*, pages 33–34. Department of Computer Science, University of Helsinki, Finland, 2018.

[60] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 3434–3440. AAAI Press, 2016.

[61] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 123–140. Springer, 2016.

[62] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Maplecomsps, maplecomsps lrb, maplecomsps chb. In *Proceedings of SAT Competition 2016 : Solver and Benchmark Descriptions*, page 52. Department of Computer Science, University of Helsinki, Finland, 2016.

[63] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 94–110. Springer, 2018.

[64] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 94–110, 2018.

[65] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Proceedings of the 2nd Israel Symposium on Theory and Computing Systems (ISTCS)*, pages 128–133. IEEE, 1993.

[66] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for cdcl sat solvers. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 703–711. AAAI Press, 2017.

[67] Inês Lynce and Joao Marques-Silva. Sat in bioinformatics : Making the case with haplotype inference. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 136–141. Springer, 2006.

[68] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, pages 62–74. Springer, 1999.

[69] Joao P Marques-Silva and Karem Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5) :506–521, 1999.

[70] Ruben Martins, Vasco Manquinho, and Inês Lynce. Improving search space splitting for parallel sat solving. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 336–343. IEEE, 2010.

[71] Ruben Martins, Vasco Manquinho, and Inês Lynce. Community-based partitioning for maxsat solving. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 182–191. Springer, 2013.

[72] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1) :165–203, 2000.

[73] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[74] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[75] H Nabeshima, K Iwanuma, and K Inoue. Glueminisat 2.2. 10 & 2.2. 10-5," 2015. *SAT Race*, 2015.

[76] Saeed Nejati and Vijay Ganesh. Cdcl (crypto) sat solvers for cryptanalysis. *arXiv preprint arXiv :2005.13415*, 2020.

[77] Saeed Nejati, Zack Newsham, Joseph Scott, Jia Hui Liang, Catherine Gebotys, Pascal Poupart, and Vijay Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 251–260. Springer, 2017.

[78] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2) :026113, 2004.

[79] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 252–268. Springer, 2014.

[80] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulae. In *Proceedings of the 2008 Conference on ECAI 2008 : 18th European Conference on Artificial Intelligence*, page 525–529, NLD, 2008. IOS Press.

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

[82] S Plaza, I Markov, and Valeria Bertacco. Low-latency sat solving on multicore processors with priority scheduling and xor partitioning. In *the 17th International Workshop on Logic and Synthesis (IWLS) at DAC*, 2008.

[83] Daniele Pretolani. Efficiency and stability of hypergraph sat algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26 :479–498, 1996.

[84] Oscar Riveros. Slime sat solver. In *Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions*, page 37. Department of Computer Science, University of Helsinki, Finland, 2021.

[85] Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.

[86] Claude. E. Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1) :59–98, 1949.

[87] Joao P Marques Silva and Karem A Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 16th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 220–227. IEEE, 1997.

[88] Tomohiro Sonobe and Mary Inaba. Counter implication restart for parallel sat solvers. In *Proceedings of the 6th International Conference on Learning and Intelligent OptimizatioN (LION)*, pages 485–490. Springer, 2012.

[89] Tomohiro Sonobe and Mary Inaba. Portfolio with block branching for parallel sat solvers. In *Proceedings of the 7th International Conference on Learning and Intelligent OptimizatioN (LION)*, pages 247–252. Springer, 2013.

[90] Tomohiro Sonobe, Shuya Kondoh, and Mary Inaba. Community branching for parallel portfolio sat solvers. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 188–196. Springer, 2014.

[91] Mate Soos. Enhanced gaussian elimination in dpll-based sat solvers. 07 2010.

[92] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 237–243. Springer, 2009.

[93] Niklas Sörensson and Niklas Een. Minisat v1.13-a sat solver with conflict-clause minimization. *International Conference on Theory and Applications of Satisfiability Testing*, 01 2005.

[94] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. P-mcomsps-str : a painless-based portfolio of maplecomsps with clause strengthening. In *Proceedings of SAT Competition 2020 : Solver and Benchmark Descriptions*, page 56. Department of Computer Science, University of Helsinki, Finland, 2020.

[95] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. Community and lbd-based clause sharing policy for parallel sat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 11–27, Cham, 2020. Springer International Publishing.

[96] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. Community and lbd-based clause sharing policy for parallel sat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 11–27, Cham, 2020. Springer International Publishing.

[97] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. On the usefulness of clause strengthening in parallel sat solving. In Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou, editors, *NASA Formal Methods*, pages 222–229, Cham, 2020. Springer International Publishing.

[98] Vincent Vallade, Ludovic Le Frioux, Razvan Oanea, Souheib Baarir, Julien Sopena, Fabrice Kordon, Saeed Nejati, and Vijay Ganesh. New concurrent and distributed painless solvers : P-mcomsps, p-mcomsps-com,p-mcomsps-mpi, and p-mcomsps-com-mpi. In *Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions*, page 40. Department of Computer Science, University of Helsinki, Finland, 2021.

[99] Allen Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 329–342, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[100] Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 116–132. Springer, 2013.

[101] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO : a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4) :543–560, 1996.

[102] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 20th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 279–285. IEEE, 2001.

[103] Xindi Zhang, Zhihan Chen, and Shaowei Cai. Parkissat : Random shuffle based and pre-processing extended parallel solvers with clause sharing. In *Proceedings of SAT Competition 2022 : Solver and Benchmark Descriptions*, page 51. Department of Computer Science, University of Helsinki, Finland, 2022.