Towards more efficient parallel SAT solving

Ludovic Le Frioux

Thesis submitted to obtain the degree of doctor of philosophy from

Sorbonne Université

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

Laboratoire de Recherche et Développement d'EPITA (LRDE)
Laboratoire d'Informatique de Paris 6 (LIP6)

Towards more efficient parallel SAT solving

–

Vers une parallélisation efficace de la résolution du problème de satisfaisabilité

Ludovic LE FRIOUX

Presented on July 3, 2019 in front of a jury composed of:

Reviewers:

- **Michaël KRAJECKI**, Professor at Université de Reims, CReSTIC
- **Laurent SIMON**, Professor at Université de Bordeaux, LaBRI, CNRS

Examiners:

- **Daniel LE BERRE**, Professor at Université d'Artois, CRIL, CNRS
- **Bertrand LE CUN**, Senior Software Engineer at Google Inc.
- **Clémence MAGNIEN**, Research Director at CNRS, LIP6, CNRS

Under the supervision of:

- **Souheib BAARIR**, Associate Professor at Université Paris Nanterre, LIP6, CNRS
- **Fabrice KORDON**, Professor at Sorbonne Université, LIP6, CNRS
- **Julien SOPENA**, Associate Professor at Sorbonne Université, LIP6, CNRS, INRIA

Contents

Remerciements

Cette partie a pour but de remercier toutes les personnes qui m'ont aidé de près ou de plus loin à accomplir ce travail. Par aider je veux dire qu'elles ont compté dans la réussite de ma mission, et que d'une manière ou d'une autre elles m'ont inspiré positivement et m'ont permis d'arriver jusqu'ici.

Je tiens à dire que même si vous n'êtes pas dans ces remerciements il ne faut pas vous formaliser, il fallait bien rédiger cette partie et c'est toujours difficile de ne pas oublier quelqu'un :)

En premier lieu j'aimerais remercier Michaël Krajecki et Laurent Simon d'avoir accepté le rôle de rapporteur de ma thèse.

Je remercie également tous les membres de mon jury d'avoir accepté de participer à celui-ci : Bertrand Le Cun, Daniel Le Berre et Clémence Magnien.

Je voudrais remercier mes encadrants de thèse : Soheib Baarir, Fabrice Kordon et Julien Sopena. Ils m'ont accompagné durant toute cette thèse et si j'en suis là aujourd'hui c'est bien grâce à eux.

Je souhaite remercier Emmanuelle Encrenaz et Laure Petrucci d'avoir fait parti de mon comité de suivi, qui m'a permis de prendre plus de recul sur mes travaux.

J'ai été initié à l'informatique et découvert la recherche grâce à eux, il est donc normal de les remercier ici : Arnaud Blanchard, Brahim Derdouri et Ghiles Mostafaoui.

Je voudrais également remercier Maxime Lorrillère, Sébastien Monnet et Julien Sopena pour les nombreux stages et projets dans lesquels ils m'ont encadré. J'ai beaucoup appris auprès de vous.

Je remercie tous mes collègues qui m'ont permis d'évoluer dans un cadre de travail plus qu'agréable aussi bien au LRDE qu'au LIP6 : Luciana Arantes, Daniela Becker, Béatrice Bérard, Marjorie Bournat, Nicolas Boutry, Antoine Blin, Jean-Baptiste Bréjon, Edwin Carlinet, Damien Carver, Joseph Chazalon, Maximilien Colange, Akim Demaille, Clément Démoulins, Alexandre Duret-Lutz, Claude Dutheillet, Jonathan Fabrizio, Arnaud Favier, Thierry Géraud, Redha Gouicem, Saalik Hatia, Le Duy Huynh, Francis Laniel, Jonathan Lejeune, Maxime Lorillère, Olivier Marin, Quentin Meunier, Laure Millet, Sébastien Monnet, Jim Newton, Minh Ôn Vũ Ngọc, Elodie Puybareau, Léo Rannou, Etienne Renault, Olivier Ricou, Julie Rivet, Michaël Roynard, Pierre Sens, Jonathan Sid-Otmane, Guillaume Tochon, Ilyas Toulmit, Yann Thierry-Mieg, Didier Verna, Gauthier Voron et Zhou Zhao. Merci à vous ! Petite mention particulière à mes co-bureau SAT : Hakan Metin et Vincent Vallade.

J’aimerais aussi remercier plusieurs personnes qui m’ont permis d’enrichir ma connaissance dans différents domaines. Concernant la logique merci à Adrien Deloro, Nouredine Lamrani et Romain Sieuzac. Merci à Stratis Limnios et Léo Rannou d’avoir partagé avec moi votre expertise sur la théorie des graphes.

Je souhaite également remercier mes amis de la fac Massinissa Bahous, Jean-Christophe Cazes, Mathieu Doz, Julie Garnier, Nouredine Lamrani, Stratis Limnios, Léo Rannou, Jonathan Sid-Otmane et Oskar Viljasaar. Petite mention particulière à mon parrain : Wandrille Domin.

Je remercie également mes amis du Parisis : Guillaume Barthélemy (Stu), Lucas Bertou (Babe), Matthieu Dumas (Bouddha), Valentine Dupart (La petite D), Antoine Guillet (Toinou), Remy Hespel (Teddy), Matthieu Kemdji (VB), Vincent Mertens (Saucisse), Charles Moussier (Rico), Cyril Petel (Pestel), Lucas Riggi (Rigui) et Tanguy Rossato (Joj).

Merci à ma famille d’avoir toujours été là avec moi : Maman, Papa, Gaëtan et Romuald.

Enfin je voudrais particulièrement remercier Camille pour son soutien quotidien, mais surtout pour tous les bons moments que l’on partage ensemble depuis une dizaine d’années. Merci aussi de m’avoir inscrit en licence d’informatique, ce n’était pas une si mauvaise idée ,p

Résumé long en français

Mots clés : parallélisation, satisfaisabilité booléenne, portfolio, diviser pour régner, échange de clauses

CONTEXTE : LE PROBLÈME DE SATISFAISABILITÉ ET SA RÉSOLUTION

Cette thèse traite de la résolution du problème de satisfaisabilité booléenne (SAT). Pour un ensemble de contraintes données (sous forme d'une formule propositielle), la résolution de SAT consiste à dire s'il existe une affectation des variables qui permet de satisfaire toutes les contraintes (*i.e.,* la formule est alors satisfaisable ou SAT) ou il n'y a aucune manière de satisfaire toutes les contraintes (*i.e.,* la formule est insatisfaisable ou UNSAT). Par exemple, considérons le problème consistant à construire un emploi du temps pour une université. Cet emploi du temps doit prendre en considération : les contraintes des professeurs, les différents types de salles (*e.g.,* amphithéâtres, laboratoires de langues, salles informatiques), les choix des différents cours qu'ont fait les étudiants, etc. Ce problème devient très vite compliqué et ingérable, par un simple individu, lorsque le nombre de contraintes augmente. Ce genre de problèmes complexes est une instance d'un problème SAT. Ce problème a été utilisé avec succès dans de nombreux contextes applicatifs incluants : la décision de planning [62], la vérification de matériel et de programme [19], la preuve automatique de théorèmes [53], ou encore la bio-informatique [75].

Le problème SAT a été le premier problème à avoir été prouvé NP-complet [27, 69]. Autrement dit on ne pense pas qu'il existe d'algorithme avec une complexité en temps mieux qu'exponentiel pour le résoudre. Malgré cela, les solveurs SAT sont en pratique très efficaces et sont capables de traiter des problèmes parfois énormes pouvant contenir des millions de variables et des milliards de contraintes (*i.e.,* clauses). En fait il existe deux classes d'algorithmes pour résoudre SAT : les algorithmes dit incomplets et ceux dit complets. Les algorithmes incomplets [94, 52, 51, 36, 25] n'offrent aucune garantie sur une éventuelle terminaison soit pour retouner une affectation satisfaisant la formule, soit pour prouver la formule comme étant insatisfaisable. Ces algorithmes sont connus pour être plus efficaces sur des instances générées aléatoirement [20, Chapitres 5 et 6]. A contrario, les algorithmes complets, qui peuvent répondre à la fois si la formule est satisfaisable ou insatisfaisable, sont plus efficaces pour traiter des problèmes issus de la vie quotidienne. Dans cette thèse nous nous intéressons uniquement à la résolution de SAT pour des formules émanant de problèmes issus de la vie quotienne et donc aux algorithmes complets.

Le premier algorithme complet (DPLL) a été proposé par Davis, Putnam, Logemann et Loveland [32, 31]. Cet algorithme a été amélioré avec l'apparition de nombreuses heuristiques et optimisations : apprentissage de clauses [81, 101, 113], heuristiques de décision [88, 70], nettoyage de la base de clauses apprises [11], structures de stockage paresseuses [89], prépocessing [40, 77, 79], redémanderages [73, 12, 16], etc.

Même si les solveurs SAT séquentiels continuent d'être de plus en plus performants, les améliorations deviennent moins significatives. D'autre part, par le passé, ces solveurs bénéficiaient de l'augmentation régulière de la fréquence des processeurs, ce n'est plus le cas aujourd'hui. De nos jours, les fabriquants de processeurs se sont tournés vers de nouvelles architectures contenant plusieurs processeurs que l'on appelle multi-coeurs.

Avec l'apparition de ces environnements parallèles (multi-coeurs, mais aussi clusters, grilles de calcul et nuages), les solveurs SAT ont dû s'adapter afin de profiter de cette nouvelle puissance de calcul. Un solveur séquentiel, que l'on peut qualifier de travailleur, utilise un processeur. Ces travailleurs sont des flux d'exécution représentés en pratique par des processus ou des *threads*. Deux grandes approches ont été étudiées afin de paralléliser les solveurs SAT :

(1) *diviser pour régner* (D&C) [112, 109, 54, 58, 7, 9, 90] consiste à diviser statiquement ou dynamiquement l'espace de recherche. Les différents sous-espaces générés peuvent alors être traités en parallèle par différents travailleurs. Si l'un des sous-espaces est rapporté comme étant SAT c'est alors toute la formule orginale qui est SAT et le calcul s'arrête. Pour montrer que la formule originale est UNSAT il faut prouver que tous les sous-espaces sont UNSAT.

(2) La seconde technique, appelée *portfolio* [49, 13, 14, 106], lance plusieurs travailleurs qui évoluent dans l'espace de recherche entier. Le premier à prouver la formule SAT ou UNSAT met fin à la résolution globale. Le but est de diversifier les travailleurs afin qu'ils explorent l'espace de recherche de manières différentes pour augmenter la probabilité de résoudre plus rapidement le problème. La diversification est souvent mise en oeuvre en utilisant différentes heuristiques ou paramètrages pour les différents travailleurs.

Dans ces environnements parallèles, les travailleurs peuvent communiquer, par le biais de la mémoire partagée lorsqu'ils sont sur la même machine, ou par l'envoi de messages sur le réseau dans le cas de machines distantes. L'échange d'informations est possible avec les deux paradigmes présentés ci-dessus (*i.e.,* D&C et portfolio). Cet échange d'informations est un point crutial du développement d'un SAT solveur parallèle. En effet échanger des informations peut accélérer la résolution, mais si trop d'informations sont échangées ou si le mécanisme d'échange est saturé alors la résolution peut être ralentie. Tout l'enjeu est alors de savoir quelles informations doivent être échangées et entre quels travailleurs.

DÉFIS IDENTIFIÉS

Malgré l'apparition de ces nouvelles solutions, le gain lié à la parallélisation de la résolution de SAT n'est pas satisfaisant et de nouvelles solutions doivent être proposées. L'étude de la bibliographie qui comprend de nombreuses contributions fait ressortir trois défis majeurs :

(D1) même si la technique `D&C` semble la plus naturelle pour paralleliser la résolution du problème SAT, les retours des différentes compétitions de solveurs SAT parallèles[1] montrent que les meilleurs solveurs sont des portfolios. **Un premier défi est donc l'amélioration des techniques utilisées dans les solveurs `D&C` afin de combler cet écart de performance avec les solveurs basés sur le paradigme de portfolio.**

(D2) L'échange d'informations entre les travailleurs est important mais reste un des points critiques de l'implémentation d'un solveur SAT parallèle. En effet une mauvaise mise en oeuvre du partage peut alors compromettre l'efficacité du solveur. Il y a deux facteurs importants à prendre en compte pour permettre un partage efficace : (1) la manière dont les informations sont echangées (avec quel mecanisme ?) et (2) quelles informations sont echangées (comment les filtrer ?). **Le deuxième défi est la mise en place de techniques et la proposition d'heuristiques pour permettre un partage d'informations plus efficace.**

(D3) Il existe de nombreuses contributions dans le domaine des solveurs SAT parallèles. Ces contributions sont en général implementées avec des bibliothèques différentes, des langages différents, et validées sur des jeux de problèmes différents. Comparer toutes ces contributions de manière équitable devient vite très compliqué. Il est donc difficile d'avoir des certitudes sur lesquelles baser sa reflexion afin de proposer de nouvelles idées. **Ainsi, le troisième défi consiste en la création d'une plateforme permettant d'implémenter les différentes techniques de l'état de l'art, ou de nouvelles, afin de pouvoir les comparer entre elles sur un pied d'égalité.**

Cette thèse s'attaque aux différents défis présentés ci-dessus et propose différentes contributions allant dans ce sens.

CONTRIBUTION 1 : PLATEFORME POUR LA RÉSOLUTION DE SAT EN PARALLÈLE

Comme la résolution des défis D1 et D2 est clairement basée sur la résolution du défi D3, celui-ci a été traité en premier. La première contribution de cette thèse est `Painless` [66], une plateforme modulaire et générique permettant l'implémentation de solveurs SAT parallèles efficaces pour des environnements multi-coeurs. Le principe de base de `Painless` est que

[1] https://www.satcompetition.org/

les aspects liés à la concurrence sont encaspulés et fournis par la plateforme, le but étant de permettre aux utilisateurs de se focaliser sur la création de stratégies qui relèvent de la résolution de SAT.

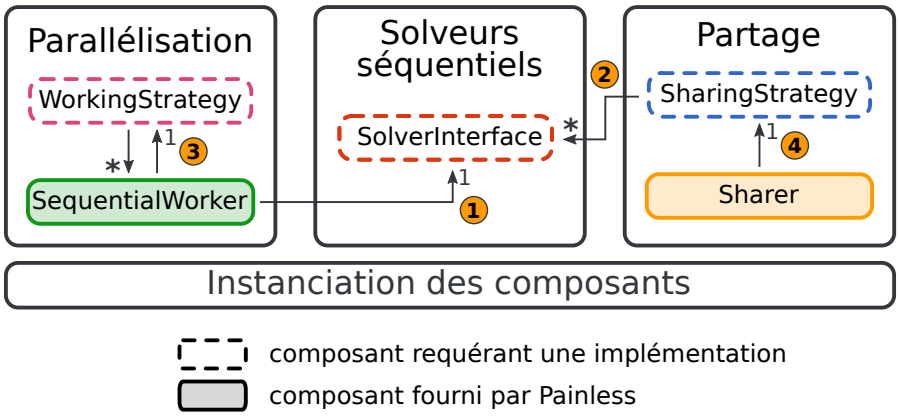


Figure 0.1: Architecture générale de `Painless`

L'architecture générale de `Painless` est présentée par la figure 0.1. Il y a trois concepts principaux dans un solveur SAT parallèle : les solveurs séquentiels sous-jacents, l'organisation des travailleurs en parallèle et le partage d'informations. L'architecture de `Painless` est divisée en trois parties, une pour chacun de ces concepts.

N'importe quel solveur SAT séquentiel peut être ajouté dans la plateforme. Afin de pouvoir les manipuler sans distinction, ces solveurs doivent être adaptés pour réspecter l'API fournie par la classe `SolverInterface`. Cette classe comporte des méthodes permettant : ❶ l'interaction avec le processus de résolution du solveur (*e.g.,* charger la formule, lancer ou interrompre la résolution); ❷ l'export/import de clauses apprises pendant la résolution. Actuellement la plateforme fournit les adapteurs pour les solveurs séquentiels suivants : `Glucose` [11], `MiniSat` [38], `Lingeling` [17] et `MapleCOMSPS` [71].

L'organisation de la parallélisation repose sur l'utilisation d'un arbre (de profondeur arbitraire) permettant la composition de stratégies. Les noeuds de l'arbre sont des instances de la classe `WorkingStrategy` encodant une stratégie de parallélisation (*e.g.,* portfolio, `D&C`). Les feuilles sont des instances de la classe `SequentialWorker` qui intègre un flux d'exécution (*thread*) opérant le solveur séquentiel associé. L'interface ❸ permet de générer facilement la descente des ordres pour résoudre le problème (`solve`) ainsi que la remontée de la terminaison (`join`). La résolution commence par un appel à la méthode `solve` de la stratégie racine et finit par un appel à la méthode `join` de cette même stratégie. Notons que la résolution commence effectivement au moment de l'appel à la méthode `solve` d'au moins une des feuilles. Actuellement la plateforme fournit les stratégies suivantes : `Portfolio`, `CubeAndConquer` [54] et un composant générique `D&C` [67] qui est décrit dans la contribution 2.

Dans `Painless` le partage est décentralisé. Chaque solveur peut exporter/importer des clauses en les mettant/prenant dans son *buffer* d'export/import. Le partage est géré par un

ou plusieurs *thread*(s) dédié(s) appelé(s) `Sharer`. Chaque `Sharer` est responsable d'une liste de producteurs et d'une liste de consomateurs. Il récupère les clauses produites par les producteurs et selon une stratégie définie par l'utilisateur (`SharingStrategy`) va les diffuser aux consomateurs. Pour se faire les stratégies doivent respecter l'API décrite par la flèche ❹ . Ce mécanisme générique est basé sur l'utilisation de *buffers lockfree* [87] qui permettent un partage efficace répondant en parti au défi D2. Actuellement la plateforme fournit les stratégies : `SimpleSharing` qui exporte toutes les clauses des producteurs vers les consomateurs, et `HordeSatSharing` qui augmente le débit de clauses apprises si celui-ci n'est pas suffisant [14].

La modularité de `Painless` a été evaluée en montrant que l'on peut implémenter de manière assez simple différents solveurs imitant les comportements de solveurs de l'état de l'art : `Syrup` [13], `Treengeling` [17] et `HordeSat` [14]. De plus l'efficacité de ces solveurs est validée par le fait qu'ils sont au moins aussi efficaces que les solveurs originaux et même mieux dans certains cas. Enfin `P-MapleCOMSPS` [65, 68] solveur crée avec `Painless` et basé sur le solveur séquentiel `MapleCOMSPS` [71] a finit 3ème et 1er lors des compétitions de solveurs SAT parallèles en 2017 et 2018, respectivement.

CONTRIBUTION 2 : IMPLÉMENTATION EFFICACE DE SOLVEURS SAT DIVISER POUR RÉGNER

Pour pouvoir avancer sur le défi D1, un composant générique permettant d'implémenter différentes techniques de `D&C` a été implémenté [67]. Ce composant repose sur l'utilisation de la plateforme `Painless` et est une implémentation de la classe `WorkingStrategy` de `Painless`.

Dans un `D&C` on divise l'espace de recherche pour résoudre les sous-espaces en parallèle. La technique la plus utilisée est le *guiding path* [112]. Pour une formule donnée φ et une de ses variables v, on peut générer grâce à la décomposition de Shannon deux sous-espaces : $\varphi \equiv (\varphi \wedge v) \vee (\varphi \wedge \neg v)$. Dans cet exemple les *guiding paths* sont : (v) et $(\neg v)$. On peut noter qu'ils sont réduits à une seule variable mais en pratique on peut en utiliser plusieurs. Le choix des bonnes variables de division est un problème difficile que l'on résout par l'emploi d'heuristiques [55, 83, 9, 7, 90].

Malgré les efforts fait pour proposer des heuristiques de division générant des sous-espaces équilibrés, il est inévitable que certains travailleurs deviennent oisifs après avoir prouver leur sous-espace UNSAT. On utilise alors des techniques afin de faire un équilibrage de charge dynamique : soit en volant du travail à un autre travailleur [7], soit en créant à l'avance des tâches qui sont stockées dans une file [54].

Au moment où un travailleur se voit attribué une tâche il y a deux manières de procéder. Soit le travailleur utilise tout au long de son existance le même objet solveur et donc il va le réutiliser pour sa nouvelle tâche (`réutilisation`). Avec l'utilisation de cette première technique, un solveur garde la connaissance séquentielle accumulée au cours des résolutions des différents

sous-espaces qui lui on été affectés. Soit on peut cloner l'objet solveur du travailleur à qui il a volé du travail (`copie`). Avec cette dernière technique il y a un surcoût dû à la copie mémoire, mais on peut espérer que le partage d'informations locale au sous-espace de recherche (fait de manière paresseuse) pourra avoir un impact positif sur la future résolution.

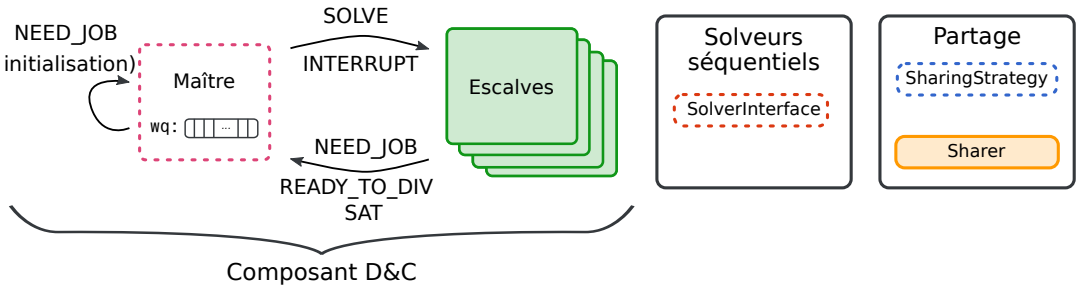


Figure 0.2: Architecture du composant diviser pour régner

L'architecture générale du composant est décrite par la figure 0.2. Le composant est basé sur une architecture maître/esclaves. Le maître envoie des évènements `SOLVE` aux différents travailleurs pour les lancer sur la résolution d'un sous-espace. Il peut aussi leur demander de s'arrêter avec un évènement `INTERRUPT`. Cette demande est asynchrone et effectuée quand un des travailleurs doit diviser son espace de recherche par exemple. Un escalve est au début oisif jusqu'à ce qu'il reçoive du travail. Il peut alors trouver que son espace est `SAT` et retouner un évènement `SAT` au maître. Ou il peut prouver que son sous-espace de recherche est `UNSAT` et donc redemander du travail (`NEED_JOB`). Si le maître lui a demandé de s'interompre, lorsqu'il est dans un état stable (*e.g.,* lorsque le solveur retourne au niveau 0 de sa résolution), il va alors notifier le maître avec un évènement `READY_TO_DIV`. A ce moment le maître pourra alors effectivement diviser le travail selon la stratégie donnée.

Grâce à ce composant il est possible d'implémenter des solveurs `D&C` qui peuvent varier sur six axes : (1) la technique pour diviser l'espace de recherche, (2) la technique pour choisir les variables de division, (3) le mécanisme d'équilibrage de charge dynamique, (4) la stratégie de partage, (5) la manière de gérer la connaissance sequentielle acquise par les travaileurs et (6) les solveurs séquentiels utilisés.

Différentes heuristiques décrites dans l'état de l'art ont été implémentées. Les six solveurs utilisés pour les expériences varient sur deux axes. Premièrement sur l'heuristique de division utilisée : VSIDS [88], nombre de flips [7] et ratio de propagations (PR) [90]. Deuxièmement nous utilisons soit une allocation de nouvelles tâches par `réutilisation`, soit par `copie`. Trois de ces solveurs sont des compositions d'heuristiques originales (n'ayant jamais été presentées dans l'état de l'art).

Les expériences nous ont permis de comparer de manière équitable les différentes heuristiques. Notamment on peut dire que l'heuristique de division basée sur le nombre de flips semble plus efficace comparée aux heuristiques basées sur le VSIDS et le PR. De plus l'allocation par `copie` est plus efficace que celle par `réutilisation`. Par ailleurs, deux des six solveurs proposés ont de meilleures performances comparés aux deux solveurs de l'état de

l'art : MapleAmpharos [90] et Treengeling [18]. En conclusion on peut dire que cette contribution est un bon avancement en vue de répondre au défi D1.

CONTRIBUTION 3 : EXPLOITATION DE LA MODULARITÉ POUR LE PARTAGE D'INFORMATIONS

Il est bien connu que les solveurs basés sur des algorithmes complets sont efficaces sur les instances issues de la vie quotidienne car celles-ci ont une certaine structure. Ceci explique que des heuristiques comme la sauvegarde de phase [95] ou le VSIDS [88] soient si efficaces.

Une manière d'exhiber cette structure est d'etudier la modularité de la formule considerée. Pour se faire la formule est transformée en graphe pondéré non orienté [3]. Dans la représentation classiquement utilisée (appelée VIG), les noeuds du graphes sont les variables de la formule et les arrêtes représentent les clauses. Il existe un lien entre deux variables si celles-ci appartiennent à une même clause. Comme ces graphes sont très gros on utilise des algorithmes afin de partitionner les noeuds (*e.g.,* Louvain [23]). Cette partition est appelée structure de communautés et chaque partie est une communauté. Cette détéction de communautés est effectuée en maximisant la modularité [91], ainsi la partition obtenue met en lumière la modulartié de la formule.

La performance du solveur MiniSat [38] pour une instance du problème SAT donnée est corrélée avec le fait que le graphe de cette formule possède une modularité forte [92]. D'autre part une des mesures clés permettant de quantifier la qualité des clauses apprises, le LBD [11], est corrélée avec le nombre de communautés représentées dans ces clauses [92].

Basé sur les précédentes corrélations, le préprocesseur modprep [4] essaie de créer de "bonnes" clauses qu'il ajoute à la formule de départ avant de la résoudre normalement. Pour se faire la formule est découpée en communautés de clauses. Chaque paires adjacentes (*i.e.,* comportant une intersection non vide de variables) est alors soumise pour résolution à un solveur SAT. Toutes les clauses apprises durant ce processus sont conservées pour étendre le problème de base. L'idée ici est de produire des clauses contenant peu de communautés et donc plutôt "bonnes" pour la résolution au vu de la corrélation entre LBD et nombre de communautés.

En pratique ce préprocessing créé peu de clauses car la plupart des sous-formules générées sont très facilement satisfaisables. En effet comme les clauses d'une même communauté ont beaucoup de variables en commun, il suffit parfois de quelques affectations pour satisfaire toute la sous-formule. On propose donc de mélanger les clauses des différentes communautés en utilisant un simple algorithme de round-robin (RR). Pour la résolution, l'algorithme utilisé est inspiré de celui présenté dans [50]. Afin de tester notre décomposition nous avons aussi proposer une décomposition aléatoire (RAND).

Sur les instances de la compétition SAT 2018 en utilisant un temps borné de 500 secondes, nous avons pu constater que nos deux préprocessings produisent plus de clauses (50945 et 55859 en valeur médiane pour RR et RAND) que modprep (219 en valeur médiane). De plus

ces clauses sont quasiment toutes de “bonnes” clauses au sens de `modprep` et contiennent rarement plus de trois communautés. Notons que les chiffres donnés ci-avant ne contiennent que ces clauses, dites “bonnes”.

Afin de voir l’impact de ces clauses sur la résolution nous avons executé `MapleCOMSPS` [71] sur les différents jeux de données (celui de base et ceux enrichis par les “bonnes” clauses générées par chacun des préprocesseurs). Les performances du solveur sont meilleures sur les instances ayant été enrichies des clauses produites par RR. On peut aussi noter que la définition de “bonnes” clauses données par `modprep` ne semble pas suffisamment restrictive car les performances du solveur sur le jeu de données enrichi des clauses de RAND sont moins bonnes que pour les autres.

Comme notre procédure est complète et qu’elle dure du coup jusqu’à ce que la résolution soit finie, nous avons incorporé ce procédé dans un solveur parallèle. Pour ce faire un des solveurs séquentiels (travailleur) est remplacé par le préprocessing qui devient du coup ici un inprocessing. Les clauses apprises par RR sont alors dispatchées aux autres solveurs en temps réel. Sur le jeu de données de la compétition 2018, les solveurs de base sont moins efficaces que ceux intégrants RR qui résolvent plus d’instances et plus rapidement.

Cette dernière contribution permet d’ouvrir des possibilités sur la manière de quantifier la qualité des clauses et permet d’avancer sur la deuxième partie du défi D2.

Abstract

Keywords: parallelisation, Boolean satisfiability, portfolio, divide-and-conquer, clause sharing

Boolean SATisfiability has been used successfully in many applicative contexts. This is due to the capability of modern SAT solvers to solve complex problems involving millions of variables. Most SAT solvers have long been sequential and based on the CDCL algorithm. The emergence of many-core machines opens new possibilities in this domain.

There are numerous parallel SAT solvers that differ by their strategies, programming languages, etc. Hence, comparing the efficiency of the theoretical approaches in a fair way is a challenging task. Moreover, the introduction of a new approach needs a deep understanding of the existing solvers' implementations.

We present `Painless`: a framework to build parallel SAT solvers for many-core environments. Thanks to its genericness and modularity, it provides the implementation of basics for parallel SAT solving. It also enables users to easily create their parallel solvers based on new strategies.

`Painless` allowed to build and test existing strategies by using different chunk of solutions present in the literature. We were able to easily mimic the behaviour of three state-of-the-art solvers by factorising many parts of their implementations. The efficiency of `Painless` was highlighted as these implementations are at least efficient as the original ones. Moreover, one of our solvers won the SAT Competition'18.

Going further, `Painless` enabled to conduct fair experiments in the context of divide-and-conquer solvers, and allowed us to highlight original compositions of strategies performing better than already known ones. Also, we were able to create and test new original strategies exploiting the modular structure of SAT instances.

Chapter

1

Introduction

Many problems can be reduced on abstracted the question of satisfying a set of constraints. For instance, consider the problem of constructing a schedule for a university. This planning should consider: professors' constraints, the different types of rooms (*e.g.,* rooms with computers, amphitheatres, language laboratories), and students' choices for different courses. This problem quickly becomes infeasible by a simple human. Fortunately, it can be handled by transforming the different constrains into an instance of the Boolean satisfiability (SAT) problem. SAT solving consists in deciding whether a propositional formula (encoding the constraints of a problem) is unsatisfiable (*i.e.,* there is no way to satisfy all the constraints), or satisfiable (*i.e.,* there is a way to satisfy all constraints).

A first reason of the success of SAT is its centrality in computer science. Also, it is the first problem that has been proven NP-complete [27, 69], this meaning that every NP problems can be solved by a transformation into SAT.

A second reason is the availability of very efficient SAT solvers [11, 71, 18]. Indeed, SAT solvers are now able to tackle huge instances involving up to millions of variables and tens of millions of constraints. Among these, we can cite instances coming from many different fields such as planning [62], hardware and software verification [19], computational biology [75], automated reasoning [53], etc. Also, SAT solvers are used as a back-end engine to solve other more complex problems such as satisfiability modulo theory (SMT) [20, Chapter 26].

There are two classes of algorithms for solving SAT problems. *Incomplete algorithms* [94, 52, 51, 36, 25] do not offer any guarantee that they will eventually either report a satisfying assignment for the formula or declare that the given formula is unsatisfiable. They are suitable for handling random instances (*i.e.,* instances generated in a random way). Contrariwise, *complete algorithms* (*i.e.,* algorithms that can answer both satisfiable and unsatisfiable) are more efficient on instances coming from real problems. This thesis focuses on complete algorithms.

For a long time, SAT solvers have been sequential, and based on the algorithm by Davis, Putnam, Logemann, and Loveland (DPLL) [32, 31]. This initial algorithm has been dramatically enhanced thanks to the introduction of sophisticated heuristics and optimisations: decision

heuristics [88, 70], clauses learning [81, 101, 113], aggressive cleaning [11], lazy data structures [89], preprocessing [40, 77, 79], etc.

Nowadays, the emergence of many-core machines and cloud computing brings on raise new possibilities in this domain. So, parallelisation is a possible axis to increase SAT solvers efficiency. Unfortunately, the adaptation of SAT algorithms towards parallelism is a complex task. Indeed, all the enhancement done in sequential tools are not always easy to parallelise. Moreover, designing efficient parallel programs require specific skills and expertises, and is a really challenging task.

This thesis treats the problem of efficiently solving SAT in parallel.

1.1 TOWARDS AN EFFICIENT PARALLELISATION

The first complete algorithm to solve SAT, DPLL, was proposed in 1962 [32, 31], and the first parallel SAT solvers was proposed in 1996 [112]. Thus during the last twenty years a very rich literature was proposed. Parallel SAT solving is a very prolific domain in terms of contributions. Mainly, two classes of parallel techniques have been developed:

- cooperation-based (*a.k.a.,* divide-and-conquer) approaches, often based on the guiding path method [112], decompose the original search space in subspaces that are solved separately by sequential solvers [54, 58, 7, 9, 90]. If one of the subspaces is reported to be satisfiable, the original formula is declared satisfiable. To prove the formula to be unsatisfiable, all the subspaces should be proven unsatisfiable.

- In the competition-based (*a.k.a.,* portfolio) setting, many sequential SAT solvers compete for the solving of the whole problem [49, 13, 14, 39]. The first one to find a solution, or proving the problem to be unsatisfiable ends the computation.

The study of the existing bibliography highlights three main challenges that should be addressed to improve parallel SAT solving:

(C1) Although divide-and-conquer approaches seem to be the natural way to parallelise SAT solving, the outcomes of the parallel track in the annual SAT competition[1] show that the best state-of-the-art parallel SAT solvers are portfolio ones. The main problem of divide-and-conquer based approaches is the search space division so that load is balanced between solvers, which is a theoretical hard problem. Since no optimal heuristics has been found, solvers compensate non-optimal space division by enabling dynamic load balancing. **A first identified challenge is a better mechanism for divide-and-conquer in order to fill the gap between divide-and-conquer and portfolio.**

[1] https://www.satcompetition.org/

(C2) With the two above-mentioned parallel strategies, information can be shared between the different underlying solvers. In practice not all learnt information can be shared between all solvers, due to hardware limitations, but also to the slowdown impact on the algorithm. Hence, more precise observations should be made to derive good heuristics. Moreover, sharing requires a particular focus when implemented because it is not necessary, it is only a bonus, and it should not slowdown the solving procedure of the different solvers. **A second identified challenge is the building and proposition of more efficient heuristics and mechanisms for sharing information.**

(C3) We have mentioned that many contributions exist in parallel SAT solving. All these contributions are difficult to compare in a fair way because they do not use the same programming language, targeted environment for execution, underlying sequential solvers, programmers' skills, instances used for experimental validation, etc. Hence, it is very complicated to plainly rely on past contributions to build future ones. Even if there are many possible good ideas, it is always difficult to move forward from ideas to efficient implementation. **A third challenge is the creation of a modular and generic framework to build and compare (new) parallel SAT solving strategies. This easiness would allow more experiments and more implementations of many different strategies, such allowing the discovery of new efficient ones.**

1.2 Contributions

This thesis contributes to the solving of the aforementioned challenges. A proposal for the first two challenges (C1, C2) clearly heavily rely on the solution we suggest for the third one (C3). So, the main outcome of this thesis is PArallel INstantiabLE Sat Solver (`Painless`), a tool simplifying the implementation and evaluation of (new) parallel SAT solvers for many-core environments. This framework aims to be a solution of the challenge C3.

The components of `Painless` can be instantiated independently to produce a complete new solver. The guiding principle is to separate the technical components dedicated to some specific aspects of concurrent programming, from the components implementing heuristics and optimisations embedded in a parallel SAT solver. This tool has been built and designed keeping in mind that we want to tackle the first and second challenges. The genericity of `Painless` has been shown with the many different strategies we were able to evaluate [13, 14, 54, 7, 90]. This is the case of parallelisation strategies, divide-and-conquer, sharing, and sequential engines. Even if it is generic and modular, we show that solvers instantiated with `Painless` are efficient. First, when mimicking state-of-the-art solvers, but also in the SAT competitions where original configurations of `Painless` were ranked 3rd and 1st respectively in 2017 and 2018.

In a first attempt to solve the challenge C1, we have extended the original version of `Painless`, by adding a modular and efficient divide-and-conquer component on top of it. The interest here is to compare in a fair way many different divide-and-conquer heuristics that

can be found in the literature. Our `Painless`-based implementation can be customised and adapted over six orthogonal mechanisms:

- (M1) technique to divide the search space;
- (M2) technique to choose the division variables;
- (M3) dynamic load balancing strategy;
- (M4) the sharing strategy;
- (M5) the workers' knowledge management;
- and (M6) the used underlying sequential solver.

Among the numerous solvers we have available, we selected some of them for performance evaluation. Thanks to our experimental results, we were able to isolate the best composition of heuristics. We also show that these solvers compete well face to natively implemented divide-and-conquer state-of-the-art solvers.

In order to quantify the quality of learnt clauses, we explored different heuristics to exploit the modularity of SAT instances. Being able to quantify the quality of clauses allows a filtering of the exchanged clauses, and hence participates to the solving of the challenge C2. Some works exist, mainly in the sequential context, that exploit this modularity of SAT instances this is implemented using graph theory and particularly community detection [23]. The engine `modprep` [4] produces "good" clauses at preprocessing to extend the original formula and speed up the solving process. We have analysed the behaviour of this mechanism to go further and produce more of these "good" clauses. To do so, we proposed an algorithm inspired from the work of [50] producing more "good" clauses. Moreover, contrariwise to `modprep` our algorithm is complete and is more interesting in the parallel context. Hence, we present experimental results with a derived inprocessing method that allows speed up of the solving. One of the classical workers is then replaced by a worker executing the algorithm producing "good" clauses.

1.3 MANUSCRIPT STRUCTURE

This manuscript is divided in 8 chapters distributed in two main parts. The first part describes and explains the context of SAT, how it can be solved in sequential, and, of course, in parallel. Second part explains the different contributions of this thesis.

The Boolean Satisfiability Problem. The goal of Chapter 2 is to better understand what is SAT, and how it can be useful. For this purpose it first introduces basics and definitions of propositional logic that will be used in the rest of this manuscript. It also defines SAT, how central it is in computer science, and explains more precisely how it can be used on some concrete examples.

Complete Sequential SAT Solving. Chapter 3 presents the complete solving of SAT in sequential. It describes the original DPLL algorithm, and the nowadays used CDCL algorithm. Different heuristics and structures that are embedded in modern efficient SAT sequential solvers are also explained. Finally, an example of the execution of CDCL is shown.

Parallel SAT Solving. Chapter 4 first describes the different environments in which a parallel SAT solver can be executed. Then it relates and classes, the more exhaustive possible, the different attempts to parallelise SAT solving that can be found in the literature.

Framework for Parallel SAT Solving. Chapter 5 introduces the `Painless` framework, and presents its general architecture. It explains how one can build different solvers while using `Painless`. Finally, it presents and discusses results of experiments we have conducted to show the modularity and effectiveness of our tool.

Implementing Efficient Divide-and-Conquer. Chapter 6 explains the mechanism of the divide-and-conquer component we have built on top of the `Painless` framework. Moreover, it presents a fair comparison of different state of the art heuristics, but also with state-of-the-art divide-and-conquer solvers. This allowed us to highlight the best combination of heuristics.

Exploiting the Modularity of SAT Instances for Information Sharing. In Chapter 7 we explore different ways to exploit the modularity of SAT instances to guide the solver. First, we give some background on modularity and graph theory. We explain how to transform SAT instances into graphs. Then we proposed a novel technique to derive "good" clauses by combining two works of the literature [4] and [50]. Finally, we show the efficiency of the approach both in sequential and parallel.

Finally, Chapter 8 concludes this manuscript and discusses different directions we have identified for future works.

Chapter

2

The Boolean Satisfiability Problem

Contents

Every work begins with a composition of elementary concepts needed to deal with the underlying problem. In this thesis, our problem consists in solving instances of the Boolean satisfiability (SAT) problem. These instances are propositional formulas representing constraint systems. Our purpose is to find, for a given constraint system, if it is possible or not to satisfy all the constraints. This is a common question and the considered instances come from many different fields such as planning decision [62], hardware and software verification [19], computational biology [75], automated reasoning [53], etc.

In this chapter, we first present basics of propositional logic (Section 2.1). Then, we define SAT and explain why it is a central problem in computer science (Section 2.2). We finally give concrete examples of the use of SAT to solve different types of real life constraint problems (Section 2.3).

2.1 PROPOSITIONAL LOGIC

The goal of logic is to define what is a correct reasoning. A reasoning is a manner to obtain conclusion from hypotheses. A correct reasoning gives nothing to the truth of the hypothesis, it only guarantees that from the truth of the hypothesis we can deduce the truth of the conclusion. The propositional logic is the logic without quantifiers, where variables can have two possible values (*true* and *false*), and allowing the construction of reasoning from the following operators: negation ($\neg$), disjunction -in other words "or"- ($\vee$), conjunction -in other words "and"- ($\wedge$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and exclusive disjunction -in other words "xor"- ($\oplus$).

2.1.1 *Syntax*

Before reasoning, we need to define the language we will use. This language is defined with the following *vocabulary*:

- constants: $\top$, and $\bot$ representing respectively true and false;
- propositional variables: a variable is an identifier, with or without index, associated with a value. For instance, valid variables can be: var, a, or x_3;
- punctuation: opening and closing parenthesis noted respectively, '(', and ')';
- operators: $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, and $\oplus$ respectively called negation, disjunction, conjunction, implication, equivalence, and exclusive disjunction.

Based on the vocabulary defined above, we now give the rules to build what we call a *strict formula*.

Definition 1: Strict Formula

A strict formula is defined inductively as follows:

- $\top$, and $\bot$ are strict formulas;
- a variable is a strict formula;
- if φ is a strict formula $\neg\varphi$ is also a strict formula;

- if φ, and ψ are strict formulas, and $\circ$ is one of the binary operators $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, and $\oplus$, then $(\varphi \circ \psi)$ is a strict formula.

Important note: In the rest of this section, we designate by $\circ$ all the binary operators defined in the vocabulary, and we simply use the term formula to designate a strict formula.

A formula can be seen as a list of symbols from the vocabulary (operators, parenthesis, variables, and constants).

The following example shows lists of symbols that are or not formulas in the sense of Definition 1.

Example: The expression $((a \wedge \neg b) \vee c)$ is a strict formula. The expressions $a \wedge \neg b \vee c$, $((a \wedge \neg b)\vee)$ and $((a \wedge \neg(b)) \vee c)$ are not strict formulas.

Definition 2: Sub-Formula

We call *sub-formula* of a formula φ, every list of consecutive symbols of φ that is a formula.

Example: Let $\varphi = ((a \wedge \neg b) \vee c)$ be a formula. $(a \wedge \neg b)$, $\neg b$, and c are some possible sub-formulas of φ. On the contrary, $\neg b) \vee c$ is not a sub-formula of φ.

With the concept of formula introduced by Definition 1, every binary operator adds a pair of parenthesis to the formula. In order to avoid too many parenthesis in formulas, and simplify formula readability, some parenthesis can be deleted to form *priority formulas*. With the deletion of parenthesis, we need to define a priority order for the different operators. Otherwise, it would become impossible to interpret these formulas. Definition 3 gives the priority order of the different operators of the vocabulary. This order is used in all the remaining of this manuscript.

Definition 3: Operators Priority Order

The priority of the different operators is given in decreasing priority order: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and exclusive disjunction ($\oplus$). For equal priority, the left operator has a higher priority, except for the implication which is right associative (*i.e.,* $a \Rightarrow b \Rightarrow c = (a \Rightarrow (b \Rightarrow c))$).

We can consider that a priority formula is the *abbreviation* of the rebuildable formula using the priorities. In the rest of this manuscript we will identify a formula and its abbreviation, and will use the term formula to designate both. In other words, what interests us in formulas, is not their writing, but their structure. The following example gives some strict formulas and their abbreviation into priority formulas.

Example: We give here multiple examples of the abbreviation of a strict formula by a priority formula:

- $a \wedge \neg b \vee c$ is the abbreviation of $((a \wedge \neg b) \vee c)$;
- $a \wedge b \wedge c$ is the abbreviation of $((a \wedge b) \wedge c)$;
- $a \Rightarrow \neg b \oplus c \Leftrightarrow d$ is the abbreviation of $((a \Rightarrow \neg b) \oplus (c \Leftrightarrow d))$.

2.1.2 *Meaning of Formulas*

We have seen, in the last section, what is a formula and how to read it. The goal of the following section is to interpret formulas.

We designate the truth values by: 0 for false, and 1 for true. The constant $\top$ is equal to 1, and the constant $\bot$ to 0. This conduct us to confound the constants and their values, and use indifferently $\top$, 1, and true; respectively $\bot$, 0 and false. The meaning of the logical operators is given in 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: Truth table of operators

In order to evaluate a formula, we give to each variable a value in the set $\mathbb{B} = \{\text{true}, \text{false}\} = \{\top, \bot\} = \{1, 0\}$. The value of the formula is then obtained by replacing the variables by their values and by applying the rules defined in Table 2.1. Although, in order to reason on formulas, we formally define here the value of a formula and present the different definitions we need.

Definition 4: Literal

A *literal* l is a variable, or its negation. For a given variable x, the positive literal is represented by x, and the negative one by $\neg x$.

Important note: For convenience, we denote $\mathcal{V}_\varphi$ ($\mathcal{L}_\varphi$) the set of variables (literals) used in φ (the index in $\mathcal{V}_\varphi$ and $\mathcal{L}_\varphi$ is usually omitted when it is clear from the context).

The value that is given to the different variables of a formula is called *assignment*. An assignment is a function as defined by Definition 5.

Definition 5: Assignment

For a given formula φ, we define an *assignment* of variables of φ, denoted α, as a function $\alpha : \mathcal{V} \to \mathbb{B}$. By abuse of notation, an assignment is often represented by the set of its true literals (*i.e.,* literals that are true for the considered assignment).

We now introduce some useful definitions linked to the concept of assignment.

Depending on whether or not all the variables of the formula have a value by an assignment, we say that this assignment is *complete* or *partial*.

Definition 6: Partial/Complete Assignment

For a given formula φ, an assignment α_p is *partial* if not all the variables of φ have an image by α_p. An assignment α_c is *complete (total)* if all the variables of φ have an image by α_c. In other words, if we consider α_p and α_c as the sets of its true literals, we have $|\alpha_p| < |\mathcal{V}|$ and $|\alpha_c| = |\mathcal{V}|$.

In order to have coherent reasoning, when manipulating assignments representing as sets of literals, we want assignments with only one possible value for each variable. Assignments respecting this constraint are said to be *consistent* as defined in Definition 7.

Definition 7: Consistent/Inconsistent Assignment

An assignment α is said to be *consistent* if and only if (iff) $l \in \alpha \Leftrightarrow \neg l \notin \alpha$, otherwise it is *inconsistent*.

In the following example, we give different assignments and precise if they are complete or partial, and consistent or inconsistent.

Example: Let $\varphi = (x \wedge y) \vee \neg z$ be a formula. Let $\alpha_1 = \{x \mapsto 0, y \mapsto 1, z \mapsto 0\}$ be an assignment of variables of φ. This assignment can also be noted $\alpha_1 = \{\neg x, y, \neg z\}$. Moreover, α_1 is complete and consistent. The assignment $\alpha_2 = \{x, \neg z\}$ is partial, because α_2 is not defined for the variable z. The assignment $\alpha_3 = \{x, y, \neg y, z\}$ is said to be inconsistent, because y has two possible values by α_3.

We have seen how variables of a given formula can have a value through assignments. We would like now to compute the value of a formula according to the value of its variables. Definition 8 gives the rules to inductively compute the value of a formula, for a particular assignment of its variables.

Definition 8: Value of a Formula

Let φ be a formula, and α an assignment of the variables of φ. $[\varphi]_\alpha$ denotes the value of the formula φ for the assignment α. The value of $[\varphi]_\alpha$ is defined recursively. Let φ and ψ be formulas, x a variable, and α an assignment, we have:

- $[\top]_\alpha = 1$, $[\bot]_\alpha = 0$;
- $[x]_\alpha = \alpha(x)$;
- $[\neg\varphi]_\alpha = 1 - [\varphi]_\alpha$;
- $[(\varphi \vee \psi)]_\alpha = max\{[\varphi]_\alpha, [\psi]_\alpha\}$;
- $[(\varphi \wedge \psi)]_\alpha = min\{[\varphi]_\alpha, [\psi]_\alpha\}$;
- $[(\varphi \Rightarrow \psi)]_\alpha =$ if $[\varphi]_\alpha$ then 1 else $[\psi]_\alpha$;
- $[(\varphi \Leftrightarrow \psi)]_\alpha =$ if $[\varphi]_\alpha$ equals $[\psi]_\alpha$ then 1 else 0;
- $[(\varphi \oplus \psi)]_\alpha =$ if $[\varphi]_\alpha$ equals $[\psi]_\alpha$ then 0 else 1.

It is clear that the value of a formula depends on the value of its variables and its structure, hence the evaluation of a formula can be presented as a *truth table*. The truth table of a given formula φ is the table representing the different possible values of φ depending on the values taken by its variables. It has individual columns for each involved variable, and a column for the corresponding value of φ. All variations of the input variables are generally listed on the left, while the output value of the formulas is usually placed in the last columns to the right.

Note that a formula φ containing n variables (*i.e.*, $|\mathcal{V}| = n$), has 2^n possible assignments, each one represented by a line in the truth table.

The truth table of the different operators of the vocabulary is presented in Table 2.1. Moreover, Table 2.2 is the truth table of the formulas: $x \wedge \neg y$, $\neg(x \Rightarrow y)$, $x \oplus x$, and $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$.

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: Example of truth table

We are now capable of evaluating the value of a formula for all the assignments of its variables. Depending of the value taken by the formula for a particular assignment, we call this assignment *model* or *counter-model* of the formula.

Definition 9: Model/Counter-Model of a Formula

For a given formula φ, if an assignment α gives the value 1 (respectively 0) to φ, *i.e.,* $[\varphi]_\alpha = 1$ (respectively $[\varphi]_\alpha = 0$), we say that the assignment α is a *model* (respectively *counter-model*) of the formula φ.

The following example presents a model and a counter-model of some formulas from Table 2.2.

Example: The assignment $\{x, \neg y\}$ is a model of the formula $x \wedge \neg y$. The assignment $\{\neg x, y\}$ is a counter-model of the formula $\neg(x \Rightarrow y)$.

Based on the value it can take, a formula can be *satisfiable* or *unsatisfiable*. Definition 10 presents these two notions. Moreover, Definition 11 introduces the concept of *tautology*.

Definition 10: Satisfiable/Unsatisfiable Formula

A formula is *satisfiable* (respectively *unsatisfiable*) if there exists at least one (respectively no) assignment of its variables that is a model. A formula is unsatisfiable if it is not satisfiable.

Definition 11: Tautology

A formula is a *tautology* if there exists no assignment that is a counter-model for the formula. In other words, the formula is true for every assignment of its variables.

The following example is also issued from the example of truth table given by Table 2.2, it presents different formulas, and identifies those that are satisfiable, unsatisfiable, or tautologies.

Example: The formulas $x \wedge \neg y$, $\neg(x \Rightarrow y)$, and $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$ are satisfiable. As shown in Table 2.2, there exists at least one assignment (line) for which the value of the formulas is 1. The formula $x \oplus x$ is unsatisfiable. As we can see in Table 2.2, for every assignment the value of the formula is 0. The formula $(x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$ is a tautology. As we can see in Table 2.2, for every assignment the value of the formula is 1.

In the remaining of this manuscript, unsatisfiable and satisfiable, will often be noted, respectively, UNSAT and SAT.

It is interesting to compare the values taken by different formulas in order to confront them. For this purpose we define here the concepts of *logical consequence*, and *logical equivalence*, respectively defined by Definition 12, and Definition 13.

Definition 12: Logical Consequence

Let φ and ψ be formulas, ψ is a logical consequence of φ if every model of φ is a model for ψ. This logical consequence is noted: $\varphi \vDash \psi$

Definition 13: Logical Equivalence

Two formulas φ and ψ are equivalent if they have the same value for every assignment. This logical equivalence is noted: $\varphi \equiv \psi$.

The following example compares formulas, presented in Table 2.2, from a logically point of view.

Example: Let $\varphi_1 = x \wedge \neg y$, and $\psi_1 = (x \Rightarrow y) \Leftrightarrow (\neg y \Rightarrow \neg x)$ be formulas. Formula φ_1 is a logical consequence of the formula ψ_1. Indeed, as we can see in Table 2.2, every model of φ_1 is a model of ψ_1. We can then write: $\varphi_1 \vDash \psi_1$. Formulas $\varphi_2 = x \wedge \neg y$ and $\psi_2 = \neg(x \Rightarrow y)$ are logically equivalent. We can see in Table 2.2, that they have the same value for every assignment. We can then write: $\varphi_2 \equiv \psi_2$.

2.1.3 *Normal Forms*

Some formulas have remarkable structural properties, we say that they are in a *normal form*. To introduce the different definitions of normal forms we are interested in, we first need to introduce the concepts of *cube*, and *clause*.

Definition 14: Cube

A cube γ is a finite conjunction of literals represented equivalently by $\omega = \bigwedge_{i=1}^{k} l_i$, or by the set of its literals $\gamma = (l_i)_{i \in [\![1,k]\!]}$.

Definition 15: Clause

A clause ω is a finite disjunction of literals represented equivalently by $\omega = \bigvee_{i=1}^{k} l_i$, or by the set of its literals $\omega = \{l_i\}_{i \in [\![1,k]\!]}$.

Some clauses have a particular notable structure. We introduced now some useful vocabulary linked to the concept of clauses, and used in the rest of the manuscript.

The first criterion to distinguish clauses is to look at their size.

Definition 16: Unit, Binary, Ternary, and n-ary Clause

A clause is a *unit*, *binary*, *ternary*, or n-*ary* if it contains respectively one, two, three, or $n \in \mathbb{N}^*$ literals.

If a clause is included in another one (when considering them as sets of literals), we say that this clause *subsumes* the second one.

Definition 17: Subsumed Clause

Let ω_1, and ω_2 be two clauses, we say that ω_1 subsumes ω_2 if all the literals of ω_1 are contained in ω_2. In other words, we have $\omega_1 \subseteq \omega_2$.

The following example presents different clauses and analyse their particularities.

Example: Clause $\omega_1 = \{\neg a\}$ is a unit. Clause $\omega_2 = \{a, \neg b\}$ is a binary. Clause $\omega_3 = \{a, b, c\}$ is a ternary. Clause $\omega_4 = \{\neg a, \neg b, \neg c, \neg e, \neg f\}$ is a 5-ary clause. Moreover, we can observe that ω_4 is subsumed by ω_1.

Formulas in normal form have remarkable structural properties. There exists many more normal forms, but we present here only two forms we are interested in for our purpose: *conjunctive normal form*, and *disjunctive normal form*.

Definition 18: Conjunctive Normal Form

A conjunctive normal form formula (CNF) φ is a finite conjunction of clauses. A CNF formula can be either represented by $\varphi = \bigwedge_{i=1}^{k} \omega_i$, or by the set of its clauses $\varphi = \{\omega_i\}_{i \in [\![1,k]\!]}$.

Definition 19: Disjunctive Normal Form

A disjunctive normal form formula (DNF) φ is a finite disjunction of cubes. A DNF formula can be either represented by $\varphi = \bigvee_{i=1}^{k} \gamma_i$, or by the set of its cubes $\varphi = \{\gamma_i\}_{i \in [\![1,k]\!]}$.

The following example illustrates normal forms and gives a CNF formula and a DNF one.

Example: The formula $\varphi = (a \vee b \vee \neg c) \wedge (\neg a) \wedge (\neg d \vee \neg b)$ is in conjunctive normal form. The formula $\psi = (a \wedge b \wedge \neg c) \vee (\neg a) \vee (\neg d \wedge \neg b)$ is in disjunctive normal form.

2.1.4 *Transforming Formulas*

There exist multiple laws that allow us to transform a formula. It is interesting, for instance, to transform any formula into an equivalent one in normal form. In fact, this goal is always reachable, as every formula can be translated into DNF, and CNF.

We now list the different laws relative to the operators $\vee$, and $\wedge$:

associativity of $\wedge$:	$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$	(2.1)
associativity of $\vee$:	$x \vee (y \vee z) \equiv (x \vee y) \vee z$	(2.2)
commutativity of $\wedge$:	$x \wedge y \equiv y \wedge x$	(2.3)
commutativity of $\vee$:	$x \vee y \equiv y \vee x$	(2.4)
distributivity of $\wedge$ over $\vee$:	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	(2.5)
distributivity of $\vee$ over $\wedge$:	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$	(2.6)
identity for $\wedge$:	$x \wedge \top \equiv x$	(2.7)
identity for $\vee$:	$x \vee \bot \equiv x$	(2.8)
annihilator for $\wedge$:	$x \wedge \bot \equiv \bot$	(2.9)
annihilator for $\vee$:	$x \vee \top \equiv \top$	(2.10)

When we consider the operator $\neg$, we add the following laws:

complementation 1:	$x \wedge \neg x \equiv \bot$	(2.11)
complementation 2:	$x \vee \neg x \equiv \top$	(2.12)
double negation:	$\neg(\neg x) \equiv x$	(2.13)
De Morgan 1:	$\neg x \vee \neg y \equiv \neg(x \wedge y)$	(2.14)
De Morgan 2:	$\neg x \wedge \neg y \equiv \neg(x \vee y)$	(2.15)

Moreover, all the other operators $\Rightarrow$, $\Leftrightarrow$, and $\oplus$ can be transformed into a formula containing only the operators $\vee$, $\wedge$, and $\neg$. We give here the transformation rules:

transformation for $\Rightarrow$:	$x \Rightarrow y \equiv \neg x \vee y$	(2.16)
transformation for $\Leftrightarrow$:	$x \Leftrightarrow y \equiv (x \Rightarrow y) \wedge (y \Rightarrow x) \equiv (\neg x \vee y) \wedge (\neg y \vee x)$	(2.17)
transformation for $\oplus$:	$x \oplus y \equiv \neg(x \Leftrightarrow y) \equiv (x \wedge \neg y) \vee (y \wedge \neg x)$	(2.18)

2.2 THE SUCCESS STORY OF SAT

The goal of this section is to define the Boolean satisfiability (SAT) problem and to understand why it has become a central problem in computer science. Section 2.2.1 introduces some background about the classification of problems regarding their difficulty. Section 2.2.2 gives a precise definition of SAT. Section 2.2.3 presents some easy cases of SAT solving (*e.g.,* 2-SAT, Horn-SAT, xor-SAT). Finally, Section 2.2.4 describes some problems that are more complex and that directly rely on SAT solving.

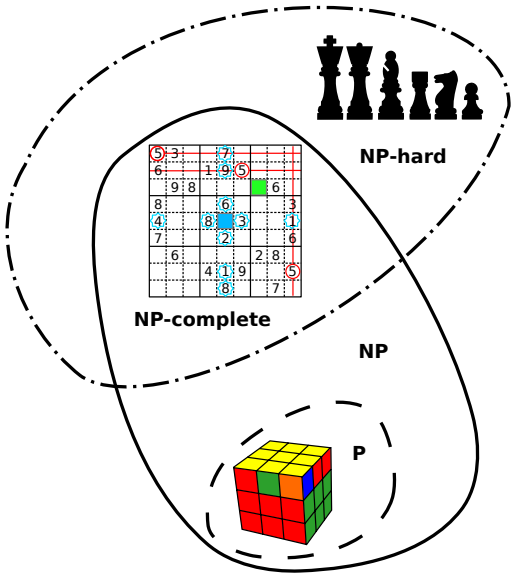


Figure 2.3: Euler diagram for P, NP, NP-complete, and NP-hard set of problems

2.2.1 *P vs. NP and the Complexity Zoo*

All the problems are not equivalent in terms of difficulty, we mean by difficulty that they do not have the same solving time complexity. On the first hand, some problems are fast to solve even when their input continues to grow. On the other hand, some have a slow solving and become not reasonably solvable when increasing the size of the input. Problems can then be classed regarding their complexity. We review in this section only some classes, but there exists many more[1].

Before introducing these classes, we define what we call a *decision problem*.

Definition 20: Decision Problem

A decision problem is a problem posed as a yes-no question of the input values. The answer to this kind of problem is then yes, or no.

Note that in the rest of this section, we consider only decision problems, and use then the term problem instead of decision problem.

The rest of this section presents different complexity classes of problems. All the classes that we present here are depicted in Figure 2.3.

First, problems that we say to be easy are part of the (Polynomial) P class of problem.

[1]There are currently 535 classes listed by the Complexity Zoo website: https://complexityzoo.uwaterloo.ca/Complexity_Zoo

Definition 21: Polynomial Problem

A problem that can be solved using a deterministic algorithm in polynomial time is part of the P (for Polynomial time) class.

These problems include for instance: multiplication, alphabetically sorting a list of words, and Rubik's cube.

Around this, including the P class, we have all the problems for which if we have an answer we are able to check it in a reasonable amount of time (*i.e.,* polynomial time). These problems are part of the (Non-deterministic Polynomial) NP class.

Definition 22: Non-deterministic Polynomial Problem

A problem that can be solved using a non-deterministic algorithm in polynomial time is part of the NP (for Non-deterministic Polynomial time) class.

These problems include for instance: sudoku, the Hamiltonian path problem, and the graph colouring problem.

Here comes one of the seven millennium prize problems "P versus NP". The question is: being able to quickly recognise correct answers to a certain problem means there is also a quick way to find these answers?

Some problems are even more complex than NP, and are part of the NP-hard class of problems.

Definition 23: NP-hard Problem

A problem p is NP-hard when every problem p' in NP can be reduced in polynomial time to p.

Chess is an example of NP-hard problem.

Finally, in the middle of this we have the NP-complete class.

Definition 24: NP-complete Problem

A problem is NP-complete if it is both NP-hard and NP.

NP-complete problems are very important in the theory of complexity because their solutions can be checked in polynomial time and, moreover, every other NP problem can be reduced into them in polynomial time. This centrality means that if for any of the NP-complete problems one finds a deterministic polynomial time algorithm that can solve it, then P will be equal to NP.

As an example the sudoku game is a NP-complete problem.

2.2.2 *Definition of the Boolean Satisfiability Problem*

In order to understand well what is SAT, we first need to define the problems of *satisfiability*, and *validity*.

Definition 25: Satisfibility Problem

The problem of satisfiability is the decision problem of determining if a given formula is satisfiable or unsatisfiable.

Definition 26: Validity Problem

The problem of validity is the decision problem of determining if a given formula is a tautology or not.

Impact of the Formula Structure. The structure of the considered formula has a huge impact on the solving of the two above-mentioned problems.
On the one hand, the satisfiability of DNF formulas is an easy problem (*i.e.,* P): one only needs to satisfy at least one cube of the formula. On the other hand, considering the problem of validity for a formula in DNF is more complicated (*i.e.,* co-NP-complete).
Contrariwise, when considering a CNF formula, verifying the satisfiability of such a formula is difficult (*i.e.,* NP-complete). While answering the question of validity for a formula in CNF is easy (*i.e.,* P), one only need to look at each clause of the formula.

We now give the definition of SAT.

Definition 27: Boolean Satisfiability Problem

The Boolean satisfiability (SAT) problem is the problem of determining the satisfiability of a given formula in CNF.

In this thesis, we are interested in solving SAT, that is very important in computer science. First of all, SAT was the first problem that was proven to be NP-complete [27, 69]. In practice, SAT solvers are able to solve huge formulas with up to millions of variables and tens of millions of clauses. Finally, the use of SAT solvers has been really simplified and harmonised by the introduction of the DIMACS input/output standard[2].

2.2.3 *Particular Cases Easy to Solve*

Some particular cases of SAT are easier to solve (*i.e.,* they are in the P class of problems). We present here the case of: *2-SAT*, *Horn-SAT*, and *xor-SAT*.

[2] https://www.satcompetition.org/2009/format-benchmarks2009.html

2-Satisfiability. The 2-satisfiability (2-SAT) problem is the problem of satisfiability for a given CNF formula containing only binary clauses.
This problem can be solved using the algorithm presented by [6]. First, the formula is transformed into *implicative normal form* that is a conjunction of implications. Using Equation 2.16 we have $(a \vee b) \equiv (a \vee b) \wedge (b \vee a) \equiv \neg a \Rightarrow b \wedge \neg b \Rightarrow a$. The translated formula is represented as a directed graph G, where nodes are literals, and each implication generates a directed edge between its two literals. We then find the strongly connected components (SCC) of G. The formula is UNSAT if the two literals of a variable (*i.e.,* positive and negative ones) are part of the same SCC. Otherwise the formula is SAT and a model can be derived. To derive the model, we sort the SCC in topological order (*i.e.,* $SCC(x) \leq SCC(y)$ if there is a path from x to y, where $SCC(l)$ returns the SCC containing the node l). The value chosen for each variable x is false if $SCC(x) < SCC(\neg x)$ and true otherwise. The complexity of thus an algorithm is linear: $O(n + m)$ where n is the number of variables and m the number of clauses.

Horn-Satisfiability. The Horn-SAT (reversed Horn-SAT) problem is the problem of satisfiability for a given CNF formula containing only Horn (reversed Horn) clauses. A Horn (reversed Horn) clause contains at most one positive (negative) literal.
This problem can be solve by applying Boolean constraint propagation (BCP), also known as unit propagation (see Section 3.1 for more details). The BCP procedure sets the values of variables in unit clauses (in cascades) in order to satisfy them. Either this procedure found the formula to be UNSAT. Or a fix point is reached and the formula is SAT. To derive a model, variables still not assigned after unit propagation are assigned to true (respectively false) for the Horn-SAT (reversed Horn-SAT) problem. Thus an algorithm has a linear complexity: $O(n + m)$ where n is the number of variables and m the number of clauses.

Xor-Satisfiability. The xor-satisfiability (xor-SAT) problem is the problem of determining the satisfiability of a given formula being a conjunction of exclusive disjunction (xor) of literals.
In order to solve xor-SAT, we can see this formula as a system of linear equations. We create a matrix A with m (number of clauses) rows and n (number of variables) columns, where every element of the matrix A_{ij} represents the presence of the variable x_j in the clause i. Since each of the clause should be satisfied the resulted column vector B of size m contains only the value 1. We pose the equation $A \times X = B$, where X is the row vector corresponding to the value of the different variables. Such a system can be solved using a classical Gaussian elimination within polynomial time complexity: $O(n^3)$.

2.2.4 *Connected Problems*

There exist satisfiability problems that are more complicated than SAT. Basically, such problems will call a SAT solver multiple times during solving.

Sharp-satisfiability Problem. The sharp-SAT (#-SAT) problem is the problem of counting the number of models of a given formula in CNF.

Maximum Satisfiability Problem. The maximum satisfiability problem (MAX-SAT) is the problem of finding the maximum subset of clauses that can be satisfied for a given CNF formula.
The weighted variant (weighted MAX-SAT) associates to each clause a weight. The goal is to find the assignments that maximize the sum of weights of the satisfied clauses.
In the partial maximum satisfiability problem (partial MAX-SAT), some clauses are said to be hard and must necessarily be satisfied. The remaining clauses (called soft) are normally considered as in the classical MAX-SAT.
Finally, there exists a variant of MAX-SAT called partial weighted MAX-SAT where the soft clauses are weighted.

Quantified Boolean Formulas. We consider here quantifiers: $\forall$, and $\exists$. The quantified Boolean formula satisfiability problem (QBF), is the problem of satisfiability for formulas in CNF with quantifiers.

2.3 Some Examples of Applications

SAT has been used successfully in many contexts such as planning decision [62], hardware and software verification [19], cryptology [84], computational biology [75], etc. This is due to the capability of modern SAT solvers to solve complex problems involving millions of variables and tens of millions of clauses.

We give in this section examples of how SAT can be used to solve concrete problems. We are first interested in the solving of two classical combinatorial problems: sudoku and n-queens puzzles. We then give an example in the verification field. Finally, we explain on an example how it is possible to use SAT to make automatic mathematical proof.

2.3.1 *Sudoku*

Sudoku is a logical based, combinatorial, number placement puzzle. The objective is to fill a $n \times n$ grid (n should be the square of an integer) with digits so that each row, each column, and each of the n sub-grids (also called regions) of size $\sqrt{n} \times \sqrt{n}$ contains all of the digits from 1 to n included. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

The classical sudoku grid size is $n = 9$. Sudoku of size $n = 4$, that are simpler to solve are often called sudokid. A more complex variant can be found with $n = 16$, this variant is called

5	**A**	*B*	*1*	*D*	**0**	*C*	*F*	*3*	**2**	**4**	*E*	**8**	*6*	**7**	**9**
8	**9**	**4**	*7*	**1**	*E*	*A*	*B*	*D*	*F*	*0*	**6**	**C**	*5*	**3**	**2**
6	*3*	*F*	*C*	*9*	*8*	**2**	*4*	*A*	*5*	**B**	**7**	*D*	**1**	*E*	**0**
2	*0*	*E*	**D**	**5**	**6**	*3*	**7**	*1*	*8*	*C*	**9**	*A*	*B*	*4*	**F**
3	**2**	*7*	*F*	*C*	**D**	*B*	*1*	*8*	**4**	**E**	*5*	*9*	*0*	*A*	*6*
B	*5*	*D*	**4**	*6*	**A**	*E*	*0*	*C*	*3*	*9*	*F*	*2*	*7*	**1**	**8**
9	*1*	**A**	*8*	*3*	*2*	*7*	**5**	**6**	*B*	*D*	*0*	*F*	*4*	*C*	*E*
0	*6*	**C**	*E*	**8**	**F**	**4**	**9**	**7**	**A**	*2*	*1*	**3**	**D**	*B*	**5**
E	*4*	*6*	*9*	**7**	*1*	**F**	*C*	*2*	*D*	**A**	**3**	*0*	**8**	*5*	*B*
F	**B**	**1**	*3*	*A*	**5**	*8*	**D**	**E**	*0*	*6*	**C**	**4**	**2**	*9*	*7*
D	**7**	**2**	*A*	*4*	**9**	**0**	**3**	**B**	*1*	*5*	*8*	**E**	*F*	**6**	*C*
C	*8*	**5**	*0*	*E*	*B*	**6**	*2*	*9*	**7**	**F**	**4**	**1**	*A*	*D*	*3*
A	**C**	**9**	**5**	*F*	*7*	*D*	*8*	**0**	**E**	**1**	*B*	*6*	*3*	*2*	*4*
7	*F*	**0**	*2*	**B**	*C*	**1**	*E*	*4*	**6**	*3*	*D*	*5*	*9*	*8*	**A**
4	*D*	*8*	*6*	*0*	*3*	*9*	*A*	**5**	**C**	*7*	*2*	**B**	*E*	*F*	**1**
1	**E**	**3**	*B*	**2**	**4**	*5*	**6**	*F*	**9**	**8**	*A*	*7*	**C**	*0*	*D*

Figure 2.4: Example of an hexadoku grid and its solution

hexadoku. An example of such a grid and its solution is depicted in Figure 2.4. The initial provided cells[3] are written in **bold black** and the solution in *italic red*.

In order to encode a sudoku grid in CNF, we need to define the notion of a *set of unordered pairs of distinct elements*.

Definition 28: Set of Unordered Pairs of Distinct Elements

Let A and B be two finite sets. A pair (a, b) is composed of $a \in A$ and $b \in B$. The set of unordered pairs of distinct elements of the sets A and B is noted $A \otimes B$ and contains all the distinct pairs (a, b). We consider that $(a, b) = (b, a)$, and if $x \in A$ and $x \in B$ then (x, x) is not in $A \otimes B$.

In order to encode the sudoku problem in CNF, we create n^3 variables $x_{i,j,k}$, where $i, j, k \in [\![1, n]\!]$. Such a variable is true if the square i, j is assigned to the digit k. The formula representing the problem of solving a given sudoku grid is $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4 \wedge \varphi_5$.

Sub-formula φ_1 represents the fact that each square has at least a value. For each square a clause of size n is generated:

$$\varphi_1 = \bigwedge_{i,j \in [\![1,n]\!]} \left(\bigvee_{k \in [\![1,n]\!]} x_{i,j,k} \right) \tag{2.19}$$

Sub-formula φ_2 represents the fact that a digit cannot be twice in a particular row. For each row, and each digit there is a set of $\frac{n \times (n-1)}{2}$ clauses generated, one per pair of squares in the row.

[3] This grid comes from: https://www.sudoku-puzzles-online.com

$$\varphi_2 = \bigwedge_{i,k\in[\![1,n]\!]} \bigwedge_{(j1,j2)\in[\![1,n]\!]\otimes[\![1,n]\!]} (\neg x_{i,j_1,k} \vee \neg x_{i,j_2,k}) \tag{2.20}$$

Sub-formula φ_3 represents the fact that a digit cannot be twice in a particular column. For each column, and each digit there is a set of $\frac{n\times(n-1)}{2}$ clauses generated, one per pair of squares in the column.

$$\varphi_3 = \bigwedge_{j,k\in[\![1,n]\!]} \bigwedge_{(i1,i2)\in[\![1,n]\!]\otimes[\![1,n]\!]} (\neg x_{i_1,j,k} \vee \neg x_{i_2,j,k}) \tag{2.21}$$

Sub-formula φ_4 represents the fact that a digit cannot be twice on a particular region. We use I to denote the row of a region and J for its column, and $I, J \in [\![1, \sqrt{n}]\!]$. We note $R(I, J)$ the set of squares of a region. For each region, and digit there is a set of $\frac{n\times(n-1)}{2}$ clauses generated, one per pair of squares in the region.

$$\varphi_4 = \bigwedge_{I,J\in[\![1,\sqrt{n}]\!]} \bigwedge_{k\in[\![1,n]\!]} \bigwedge_{((i1,j1),(i2,j2))\in R(I,J)\otimes R(I,J)} (\neg x_{i_1,j_1,k} \vee \neg x_{i_2,j_2,k}) \tag{2.22}$$

Sub-formula φ_5 represents the digits that are already provided by the setter. For each digit k placed in the square i, j, the unit clause $\{x_{i,j,k}\}$ is created and added to φ_5.

When the formula is created, it can be given to a SAT solver that will find the solution of the sudoku grid. In the model returned by the solver, a variable ($x_{i,j,k}$) with a value at true means that the square i, j can be filled with the digit k. If in the model the value of a variable is false, it can be ignored.

The solution of the hexadoku depicted in Figure 2.4 has been computed by encoding the problem into a CNF formula and solving it with the help of a SAT solver.

2.3.2 n-*Queens Puzzle*

The n-queens puzzle is the problem of placing n chess queens on a $n \times n$ chessboard so that two queens never threaten each other. In other words, no two queens can be on the same row, column, or diagonal.

In order to encode the n-queens problem in CNF, we need to create n^2 variables $x_{i,j}$, where $i, j \in [\![1, n]\!]$. Such a variable is true if there is a queen on the case i, j. The formula representing the problem is $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$.

Sub-formula φ_1 represents the fact that each line can contain at most one queen, and at least one:

$$\varphi_1 = \bigwedge_{i\in[\![1,n]\!]} \Big(\Big(\bigwedge_{(j1,j2)\in[\![1,n]\!]\otimes[\![1,n]\!]} (\neg x_{i,j1} \vee \neg x_{i,j2})\Big) \wedge \Big(\bigvee_{j\in[\![1,n]\!]} x_{i,j}\Big)\Big) \tag{2.23}$$

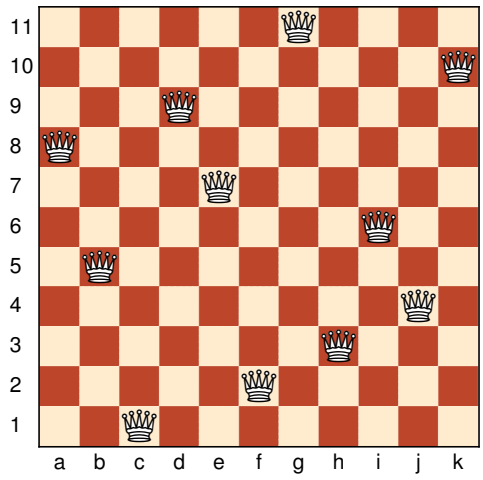


Figure 2.5: Example of solution for the 11-queens problem

Sub-formula φ_2 represents the fact that each column can contain at most one queen, and at least one:

$$\varphi_2 = \bigwedge_{j \in [\![1,n]\!]} ((\bigwedge_{(i1,i2) \in [\![1,n]\!] \otimes [\![1,n]\!]} (\neg x_{i1,j} \vee \neg x_{i2,j})) \wedge (\bigvee_{i \in [\![1,n]\!]} x_{i,j})) \tag{2.24}$$

Sub-formula φ_3 represents the fact that each diagonal can contain at most one queen. In order to define this formula, we first define $\mathcal{D}_n$, the set of diagonals of size $s > 1$. Each diagonal is a set of pair (i,j) indicating that the square (i,j) is part of the diagonal. For instance: $\mathcal{D}_2 = \{\{(1,1),(2,2)\},\{(1,2),(2,1)\}\}$.

$$\varphi_3 = \bigwedge_{d \in \mathcal{D}_n} (\bigwedge_{((i1,j1),(i2,j2)) \in d \otimes d} (\neg x_{i1,j1} \vee \neg x_{i2,j2})) \tag{2.25}$$

Using the encoding given above, we solve the n-queens puzzle with $n = 11$. A possible solution is depicted in Figure 2.5.

2.3.3 *Bounded Model Checking*

Let $\mathcal{M}$ be a model represented as an automaton and $\mathcal{P}$ be a property. The bounded model checking (BMC) [19] is the action of verifying if within k transitions there exists an accessible state of $\mathcal{M}$ that does not satisfy $\mathcal{P}$.

We present here an example based on the modelling of a two-bit counter. The automaton describing the different states of such a counter is depicted in Figure 2.6. We want to verify if there is a reachable state in $k = 4$ transitions that does not respect the following property $\mathcal{P}$:

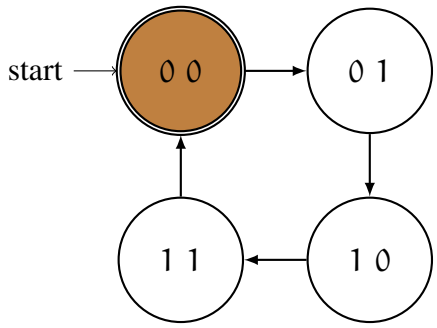


Figure 2.6: Automaton describing the behaviour of a two-bit counter

"*at least one of the two bits of the counter is equal to zero*". We then create the formula φ_k as follows:

$$\varphi_k = I \wedge T_k \wedge \neg P_k \tag{2.26}$$

For each state $i \in [\![1,4]\!]$ we generate a pair of variables l_i and r_i representing the two bits of the counter. Variable l_i (r_i) represents the left (right) bit of the counter. Such a variable is true (false) if the corresponding bit is equal to 1 (0).

Formula I encodes the initial configuration of the system. The counter is initialised to 0, we have then for every value of k:

$$I = \neg l_0 \wedge \neg r_0 \tag{2.27}$$

In a general manner, formula T_k encodes the transition relation of the system, with at most k steps from the initial state. For our example we have:

$$T_k = (\bigwedge_{i=0}^{k-1} l_{i+1} \Leftrightarrow \neg(l_i \Leftrightarrow r_i) \wedge r_i \Leftrightarrow r_{i+1}) \tag{2.28}$$

Using Equation 2.17, we can replace the equivalences:

$$\begin{aligned} T_k = \bigwedge_{i=0}^{k-1} ((\neg l_{i+1} \vee \neg((\neg l_i \vee r_i) \wedge (l_i \vee \neg r_i))) \wedge \\ (l_{i+1} \vee (\neg l_i \vee r_i) \wedge (l_i \vee \neg r_i)) \wedge \\ (\neg r_i \vee r_{i+1}) \wedge (r_i \vee \neg r_{i+1})) \end{aligned} \tag{2.29}$$

Using De Morgan's laws given by Equations 2.14 and

2.15, we obtain:

$$T_k = \bigwedge_{i=0}^{k-1}((\neg l_{i+1} \vee (l_i \wedge \neg r_i) \vee (\neg l_i \wedge r_i))\wedge (l_{i+1} \vee (\neg l_i \vee r_i) \wedge (l_i \vee \neg r_i))\wedge (\neg r_i \vee r_{i+1}) \wedge (r_i \vee \neg r_{i+1})) \tag{2.30}$$

Using Equation 2.6, we get:

$$T_k = \bigwedge_{i=0}^{k-1}((\neg l_{i+1} \vee \neg l_i \vee \neg r_i) \wedge (\neg l_{i+1} \vee l_i \vee r_i) \wedge (l_{i+1} \vee \neg l_i \vee r_i)\wedge (l_{i+1} \vee l_i \vee \neg r_i) \wedge (\neg r_i \vee r_{i+1}) \wedge (r_i \vee \neg r_{i+1})) \tag{2.31}$$

Formula P_k encodes property $\mathcal{P}$ for the states reachable within at most k transitions. For our example, we have:

$$\neg P_k = (\bigvee_{i=0}^{k-1} \neg(\neg l_i \vee \neg r_i)) \tag{2.32}$$

Using Equation 2.15, we obtain:

$$\neg P_k = (\bigvee_{i=0}^{k-1} l_i \wedge r_i) \tag{2.33}$$

Finally, we can develop the formula using Equation 2.6. We call here $\mathcal{S}_k$ the combinatorial set of all the clauses of size k that can be generated using the literals l_i and r_i with $i \in [\![0, k-1]\!]$. We obtain:

$$\neg P_k = \bigwedge_{\omega \in \mathcal{S}_k} \omega \tag{2.34}$$

When we encode the problem and solve it using a SAT solver, we have that φ_2 is unsatisfiable, meaning that there are no states reachable in two steps that do not respect the property $\mathcal{P}$. The formula φ_3 is satisfiable, the returned model represents a counter-example of an execution that leads to the violation of the property $\mathcal{P}$, indeed the state where the counter value is 3 does not respect the property and is reachable in 3 steps.

2.3.4 *Boolean Pythagorean Triples Problem*

The *Boolean Pythagorean triples problem* is a problem coming from Ramsey's theory. It is the decision problem asking if the set of natural numbers ($\mathbb{N}$) can be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$, and $a \neq b \neq c$.

In order to encode this problem in CNF, we consider the problem for $a, b, c \in [\![1, n]\!]$, and create n variables x_i. Such a variable is true (false) if the number i is part of the first (second) part. We note $\mathcal{T}_n$ the set of triples $(a, b, c) \in [\![1, n]\!]^3$ where $a \neq b \neq c$ and $a^2 + b^2 = c^2$. This set of triples is identified offline. For each triple we create a sub-formula $\varphi_{a,b,c}$. The global formula φ is then the conjunction of all the sub-formulas.

$$\varphi = \bigwedge_{(a,b,c)\in\mathcal{T}_n} \varphi_{a,b,c} \tag{2.35}$$

For each identified Pythagorean triple (a, b, c), we do not want all three variables (x_a, x_b, x_c) to be equal (*i.e.,* not to be in the same part). We have:

$$\varphi_{a,b,c} = \neg((x_a \Leftrightarrow x_b) \wedge (x_a \Leftrightarrow x_c) \wedge (x_b \Leftrightarrow x_c)) \tag{2.36}$$

Using Equation 2.17 giving the transformation of equivalence into implications, we obtain:

$$\varphi_{a,b,c} = \neg((x_a \Rightarrow x_b) \wedge (x_b \Rightarrow x_a) \wedge (x_a \Rightarrow x_c) \wedge (x_c \Rightarrow x_a) \wedge (x_b \Rightarrow x_c) \wedge (x_c \Rightarrow x_b)) \tag{2.37}$$

Using Equation 2.16 giving the transformation of implication into negations, conjunctions and disjunctions, we obtain:

$$\varphi_{a,b,c} = \neg((\neg x_a \vee x_b) \wedge (\neg x_b \vee x_a) \wedge (\neg x_a \vee x_c) \wedge (\neg x_c \vee x_a) \wedge (\neg x_b \vee x_c) \wedge (\neg x_c \vee x_b)) \tag{2.38}$$

Using the first Morgan law given by Equation 2.14, we obtain:

$$\varphi_{a,b,c} = (x_a \wedge x_b) \vee (\neg x_b \wedge x_a) \vee (\neg x_a \wedge x_c) \vee (\neg x_c \wedge x_a) \vee (\neg x_b \wedge x_c) \vee (\neg x_c \wedge x_b) \tag{2.39}$$

Using the distributivity of $\wedge$ over $\vee$ given by Equation 2.5, we obtain:

$$\varphi_{a,b,c} = (x_a \vee x_b \vee x_c) \wedge (\neg x_a \vee \neg x_b \neg x_c) \tag{2.40}$$

We encode in CNF the problem for $n = 100$ using the general rules given above. By analysing the model providing by a SAT solver, we construct the results presented in Table 2.7. Each

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Table 2.7: One solution for the Boolean Pythagorean triples problem with $n = 100$

square represent a number, if it is coloured in blue (red) then the number should be placed in the first (second) part. If the square is white, this means that the number is not part of a Pythagorean triple and can be put in one of the two parts without restriction, in other words, it is a don't care variable.

Note that it has been shown by [53], that the answer to this problem is no. Hence, the problem is satisfiable until $n = 7824$, and become unsatisfiable for $n = 7825$. Moreover, we can note that the proof produced by the used SAT solver for the unsatisfiability of the problem with $n = 7825$, is the bigest proof ever generated with a size of 200TB [4].

2.4 SUMMARY AND DISCUSSION

We have presented in this chapter, the propositional logic without quantifiers, SAT, and some of its possible applications. In the rest, of this manuscript we will only be interested in CNF formulas, hence, we will often use the term formula to designate a CNF formula.

Our purpose in this thesis is to improve the efficiency of SAT solvers, software programs capable of solving SAT considering the formula given as input. By efficiency we mean that we want to implement solvers solving more instances and the fastest possible.

We can also notice that the applications given as examples are typical possible formulas that are commonly used to evaluate the different implementations of SAT solvers. For instance, benchmarks from the different SAT competitions / races / challenges[5] contain instances coming from many fields. We will never make any assumption on the origin of formulas and will solve all of them in the same way. These instances will be used to validate (or not) our different implementations of solvers.

[4] https://www.cs.utexas.edu/~marijn/ptn/
[5] https://satcompetition.org

Chapter

3

Complete Sequential SAT Solving

Contents

This section surveys complete sequential algorithms to solve SAT formulas. We do not deal with incomplete algorithms [94, 52, 51, 36, 25], which do not provide a guarantee that they will eventually either report a satisfying assignment or prove the formula to be UNSAT. Complete SAT algorithms are more adapted to solve structured instances coming from real life problems, while incomplete ones are more efficient on random instances.

Section 3.1 presents the DPLL algorithm [32, 31]: it is the first algorithm that has been proposed to solve SAT. Therefore, Section 3.2 introduces the CDCL algorithm [81, 101, 113] used nowadays in all modern sequential SAT solvers. The DPLL-based algorithm owes its effectiveness to clause learning and conflict analysis, which are discussed in Section 3.3. Section 3.4 explains different branching heuristics that can be found in complete sequential SAT solvers. Section 3.5 presents some optimisations that are classically embedded in nowadays SAT solvers. Section 3.6 focuses on preprocessing and inprocessing mechanisms that can help to speed up solving. An example of the operation of CDCL is depicted in Section 3.7. Finally, Section 3.8 concludes this chapter.

3.1 PRINCIPLES OF THE BASIC RESOLUTION ALGORITHM

Algorithm 1 presents the algorithm developed by Davis, Putnam, Logemann, and Loveland (DPLL) [32, 31]. Its objective is to determine whether a formula φ, given as input, is SAT or UNSAT. It is a depth-first search over a binary tree where nodes are derived formulas. A leaf corresponds to one of the two following situations:

(1) the derived formula contains an empty clause, showing the current branch is not a model of φ (Line 4);

(2) the derived formula is empty, indicating the current branch is a model of φ (Line 7).

When all the leaves of the binary search tree correspond to case (1), φ is UNSAT. If at least one leaf corresponds to case (2), φ is SAT. It is then possible to display the found model by tracking $\mathfrak{x}$ over the recursive calls.

The construction of the binary search tree relies on `unitPropagation` (see Algorithm 2), `purePropagation` (see Algorithm 3), and `decisionHeuristic` functions.

Note that the `orelse` operator (used at Line 9) acts like an `or`, but as soon as one condition is satisfied, the others are not checked. Conditions are checked from left to right.

The `unitPropagation` function sets the values of variables in unit clauses in order to satisfy them (Line 3). This procedure is applied until a fixed point is reached: either there are no more unit clauses in the formula, or an empty clause has been created (Line 2). In the latter case, the current assignment reaches a leaf of type (1) and the search backtracks.

```
1 function DPLL(φ: a CNF formula)
    /* returns ⊤ if φ is SAT else ⊥                                    */
2   φ ← unitPropagation(φ)
3   if {} ∈ φ then
4     return ⊥                          // φ contains an empty clause and is UNSAT
5   φ ← purePropagation(φ)
6   if φ = ∅ then
7     return ⊤                                          // φ is empty and is SAT
8   x ← decisionHeuristic(φ)
9   return DPLL(φ|x) or else DPLL(φ|¬x)
```

Algorithm 1: Davis, Putnam, Logemann, and Loveland (DPLL)

```
1 function unitPropagation(φ: CNF formula)
    /* return CNF formula                                              */
2   while ∃{l} ∈ φ and {} ∉ φ do
3     φ ← φ|l                  // Delete all clauses containing l, and all literals ¬l
4   return φ
```

Algorithm 2: Unit propagation

The `purePropagation` function looks in φ for variables that appear as positive or negative literals but not both at the same time (Line 2). The value of such a variable is chosen such that it makes this literal true. Finally, all clauses containing this variable become satisfied and can be deleted from the formula (Line 3). In order to apply pure propagation, all the literals of all clauses must be visited. Pure propagation is not possible if only part of the literals from each clause are stored over the recursive calls as is the case in modern SAT solvers. This type of storage is known as *lazy data structure* and is explained in Section 3.5.4.

```
1 function purePropagation(φ: CNF formula)
    /* returns CNF formula                                             */
2   while ∃l ∈ L_φ s.t. ¬l ∉ L_φ do
3     φ ← φ|l                          // Delete all clauses containing l in φ
4   return φ
```

Algorithm 3: Pure propagation

The `decisionHeuristic` function is the most crucial part of the algorithm from an efficiency point of view. The objective is to find the variable that will generate a maximum of unit propagations. However, since finding this optimal variable is NP-Hard [20], research focuses on defining heuristics [60, 26, 37, 42, 97]. Section 3.4 deals with the most commonly used heuristics in modern state-of-the-art solvers.

3.2 PRINCIPLES OF OPTIMIZED RESOLUTION

The main problem of the DPLL algorithm is that it does not learn from its errors. Therefore, it can encounter the same errors several times, incurring unnecessary CPU time usage. Hence, the main lead identified by the research community to improve upon the algorithm is to propose *learning schemes* [81, 101, 113]. *Conflict driven clause learning* (CDCL) implements this concept.

Algorithm 4 gives an overview of CDCL. Like DPLL, it walks a binary search tree. The algorithm is based on a main loop that first applies `unitPropagation`[1] on the formula φ for the current assignment α (Line 5). If the formula is empty, the algorithm returns $\top$ (Line 8), and α contains the resulting model. If the formula contains an empty clause, then two scenarios are possible: (i) we are at level 0 and the algorithm returns $\bot$ (Line 11); (ii) otherwise we deduce the reasons for the empty clause and a backjump point is computed (Lines 12-15). Else, a new literal is selected to progress in the resolution of φ (Lines 17-18). This algorithm relies on the following functions: `conflictAnalysis` and `backjumpAndRestart`. These functions are explained thereafter in Section 3.3 and Section 3.5 respectively. Note that `dl` represents the number of decisions in the current branch, often called *decision level*.

3.3 CONFLICT ANALYSIS

There is a conflict when the resolution algorithm encounters a situation requiring a variable to be simultaneously set to $\top$, and to $\bot$ (*i.e.,* leaf of type (1)). As an example, let $\varphi_1 = \{\{x_1, x_2\}, \{x_4, x_5\}, \{x_4, x_6\}, \{\neg x_5, \neg x_6, x_7\}, \{\neg x_7, \neg x_8\}, \{\neg x_2, x_3, x_8\}\}$ be a formula, and $\alpha = \{\neg x_1, x_2, \neg x_3, \neg x_4, x_5, x_6, x_7\}$ the current assignment, we have $\varphi_1|_\alpha = \{\{x_8\}, \{\neg x_8\}\}$. Thus x_8 must be equal to $\top$ and $\bot$ to satisfy $\varphi_1|_\alpha$, leading to a conflict.

By making the same decisions/propagations again, a given conflict may occur several times during the resolution. To avoid this repetition, the reasons for a conflict should be identified and leveraged to guide latter search. This is done by building and analysing what we call the *implication graph*.

3.3.1 *The Implication Graph*

The implication graph is a representation of the current state of the proof system. It is updated every time a variable is assigned (by decision or propagation) or unassigned (by a backjump or a restart). Abbreviated as IG, it is a directed acyclic graph where vertices represent assignments, and edges the reasons for these assignments. Let φ be a formula, α an assignment, and $\rho : \alpha \to \varphi$ the reasons for these assignments. The IG is defined by $\mathrm{IG} = (\alpha, E, \rho)$ where

[1] Note that it is exactly the same procedure as the one used for DPLL. Since we are now using an iterative algorithm, we have to keep track of the current assignment. We add a second return parameter to this function (a list of the literals propagated during the function call).

```
1  function CDCL(φ: CNF formula)
       /* returns ⊤ if φ is SAT else ⊥                                   */
2      α ← ∅                                              // Current assignment
3      dl ← 0                                             // Current decision level
4      forever
5          (φ′, α′) ← unitPropagation(φ|α)
6          α ← α ∪ α′                                     // Add propagated literals in α
7          if φ′ = ∅ then
8              return ⊤                                   // φ is SAT
9          else if {} ∈ φ′ then                           // There is a conflict to be analysed
10             if dl = 0 then
11                 return ⊥                               // φ is UNSAT
12             ω ← conflictAnalysis(φ, α)
13             φ ← φ ∪ {ω}
14             dl ← backjumpAndRestart(dl, ω, ...)
15             α ← {l ∈ α | δ(l) ≤ dl}
16         else
17             α ← α ∪ {decisionHeuristic(...)}           // Pick a new decision literal
18             dl ← dl + 1
```

Algorithm 4: Conflict driven clause learning (CDCL)

$E = \{(l_1, l_2) | (l_1, l_2) \in \alpha^2, \neg l_1 \in \rho(l_2)\}$. Each vertex of an IG is labelled $l@\delta(l)$, where $l \in \alpha$, and $\delta(l)$ is the decision level of the assignment l. An edge $l_1@\delta(l_1) \xrightarrow{\omega} l_2@\delta(l_2)$ means that assignment l_1 implies by unit propagation the assignment of l_2 using the clause ω. We note the reason for the assignment of l_2: $\rho(l_2) = \omega$. As decision are not implied by propagation (*i.e.,* they are arbitrary choices), nodes corresponding these assignments have no incoming edges. The IG is defined by $IG = (\alpha, E, \rho)$ where $E = \{(l_1, l_2) | (l_1, l_2) \in \alpha^2, \neg l_1 \in \rho(l_2)\}$.

Figure 3.1 shows the IG corresponding to the conflict reached during the resolution of formula φ_1. There are three decisions taken in the following order: $\neg x_1$ (level 1), $\neg x_3$ (level 2), and $\neg x_4$ (level 3). Moreover at level 1, x_2 has been deduced using ω_1; and at level 3 x_5, x_6, x_7, x_8, and $\neg x_8$ have been propagated using respectively ω_2, ω_3, ω_4, ω_5, and ω_6. These last assignments of x_8 to $\top$, and x_8 to $\bot$ lead to a conflict.

3.3.2 *Computing the Reasons for a Conflict*

As unit propagation is sequential, there can only be one conflict at a given time. The IG representing the current state of the proof system containing the conflict is analysed.

Nodes called *unique implication points* (UIP) in an IG are nodes representing the convergence in the implication system and are useful to determine the origin of conflicts. Let a and b be

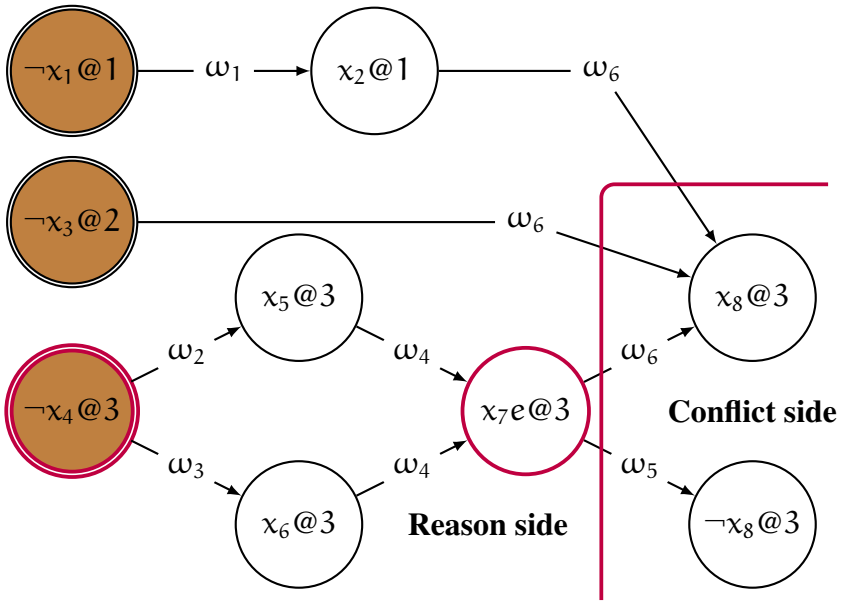


Figure 3.1: Example of implication graph and its use

: decision assignment — : implication — : unique implication point
: deduced assignment — : conflict — : cut

nodes of an IG belonging to the conflict's decision level. We say that a *dominates* b iff every path from the decision variable of the current decision level to b go through a. An UIP is a node from the current decision level that dominates the nodes implicated in the conflict. There is at least one UIP because the decision of the conflict's decision level is by definition an UIP.

On the IG of Figure 3.1 there are two UIPs: $\neg x_4@3$, and $x_7@3$. Note that $x_7@3$ is the first UIP (1^{st}-UIP), which is closest to the conflict. It is well known that the 1^{st}-UIP provides the smallest set of assignments that are responsible for the conflict [81]. These assignments are said to be reasons for the conflict.

An UIP divides the IG in two sides. The conflict side groups the two assignments involved in the conflict and the assignments located after the UIP. The reason side contains all the remaining assignments. Each node with an outgoing edge from the reason side to the conflict side is a cause of the conflict. Applying the algorithm to our example, the cube $\{x_2, \neg x_3, x_7\}$ is identified as the origin of the conflict (see Figure 3.1).

3.3.3 *Learning from a Conflict*

The following scheme is used to produce a *learnt clause* whose purpose is to avoid encountering the same conflict again. If $\bigwedge_{i=1}^{k} l_i$ is the origin of a conflict, then $\bigvee_{i=1}^{k} \neg l_i$ is a necessary condition to avoid this conflict. Since this information is already contained in the original formula, this *learnt clause* can safely be added to the original formula, and removed latter. In our example, the following learnt clause is obtained: $\omega = \{\neg x_2, x_3, \neg x_7\}$, and $\varphi_1 \equiv \varphi_1 \wedge \omega$.

3.4 Branching Heuristics

This section presents the principles used to define branching heuristics for guiding the exploration of the search tree, thus accelerating computation.

3.4.1 *Decision Heuristics*

The complexity of the resolution of SAT is strongly related to the depth of the search tree. As its construction is dominated by the choice of decision variables, choosing *good* decision variables (*e.g.,* those that generate the most unit propagations) considerably decreases computation time. Numerous heuristics have been defined to increase propagation, thus reducing the depth of the search tree [26, 37, 42, 97, 60, 80]. Recent works deal with variables involved in the most recent conflicts.

Variable State Independent Decaying Sum. VSIDS has been proposed by [88], and is used today in almost all SAT solvers. For each variable, a score is computed during the search: at each conflict, the variables involved in learnt clauses and those used in the resolution of the conflict are rewarded. When a decision is needed, the unassigned variable with the highest score is chosen. All scores are periodically multiplied by a factor $\alpha \in [0, 1]$ in order to avoid overflows. VSIDS scores are then very volatile.

Learning Rate Branching. LRB heuristic has been introduced by [70], it is a generalisation of VSIDS. The decision choice process is seen as an optimisation problem where the objective is to maximise the *learning rate* (LR), defined as the ability of variables to generate learnt clauses. This heuristic is based on *reinforcement learning*[2]. Let $\mathcal{I}$ be the interval of time from the assignment of a variable v until v is unassigned. $L(\mathcal{I})$ is the number of clauses learnt during $\mathcal{I}$. Let $P(v, \mathcal{I})$ be the number of learnt clauses in which v participated (*i.e.,* v is in the clause or v is in the conflict size of the IG) during $\mathcal{I}$. Then, LR is defined by $\frac{P(v,\mathcal{I})}{L(\mathcal{I})}$. The LRB of a variable is the average value taken by its LR over time, more recent LR are preferred to older ones (*i.e.,* they have a higher weight). The algorithm used here is *exponential recency weighted average* (ERWA). Moreover, LRB uses two more extensions: reason side rate and locality, see [70] for more details.

3.4.2 *Polarity Heuristics*

When a decision variable is chosen, its *polarity* ($\top$ or $\bot$) remains undetermined. The strategy implemented in almost all solvers is *phase saving* (or *progress saving*) [95]. As it has

[2]Reinforcement learning is a machine learning discipline, where at each turn agents choose to trigger an action among several provided by the environment. These actions are chosen in order to maximise some gain function. We refer the reader to [108] for more details.

been pointed out in [21], industrial instances contain components. A component is a subset of clauses, two components are independent if they share no variables. When a conflict is reached, the search can backjump other multiple levels (see Section 3.5.1). The assignment of some variables are forgotten, this may lead to the loss of some components resolutions. In order to counter this phenomenon, the last assignment of variables forced by unit propagation is stored. When a decision variable is chosen, it is assigned to the value stored in the phase. Moreover, this technique forces the solver to continue exploring the same zone, and *acts as a cache storing variables' hot values*.

3.5 OPTIMISING THE EXPLORATION

This section presents different optimisation that are used in SAT solvers to accelerate the computation.

3.5.1 *Backjumping*

In the DPLL algorithm, when a conflict is detected, a backtrack (to the previous decision level) is performed. Now, conflict analysis enables to compute backjumps, that can be longer than simple backtracks, and represent potentially better checkpoints to continue the search for a solution. Actually, we must backjump to the highest level that lets the learnt clause assertive (Line 5 in Algorithm 5).

```
1 function backjumpAndRestart(dl: current level, ω: learnt clause, ...)
    /* returns backjump level                                          */
2   if restartPolicy(...) then
3     return 0
4   else
5     return max{δ(l) | ¬l ∈ ω and δ(l) ≠ dl}
```

Algorithm 5: Backjump and restart policies

3.5.2 *Restarting*

Even if it backjumps and learns clauses, it happens that the solver stalls at the same search zone for a long time (it does not make significant progress). The phenomenon is known as *heavy tailing* [20]. A way to escape from this zone is to restart the search, while keeping useful information (*e.g.,* learnt clauses). Technically, a restart is a backjump to level 0 (Line 3 of Algorithm 5). Detecting a heavy tailing area is not an easy problem and has been widely treated in the literature (for example, see [73, 12, 16]). These strategies can be based on counting the number of conflicts or on the monitoring of the current search's depth.

3.5.3 *Cleaning Learnt Clauses Database*

Too many clauses are learnt during the search, thus leading to a storage problem and slowing down the unit propagation mechanism. To avoid this, we must select the most relevant clauses to guide the future search. However, defining the relevance of a clause is a hard problem that can be only solved using heuristics:

Size. The *size* of the clauses is used in many solvers. A clause with few literals needs few assignments before it becomes unit and can be used for propagation. Moreover, a larger clause has more chance to become satisfied (no more possible propagation due to this clause).

Clause Activity. The more often clauses are used for propagation and resolution of conflicts the higher *activities* they have. A clause with a high activity is often used and should be kept. This measure is used in many solvers (*e.g.,* `MiniSat` [38]).

Literal Block Distance. The LBD measure was introduced in [11], and implemented in the `Glucose` [11] solver. The LBD of a learnt clause is the number of decision levels represented by its variables (lower is better). In particular, clauses that have LBD value of two (called *glue clause*) are very important because they allow to merge two decision levels. This measure is now used in almost all modern competitive solvers.

3.5.4 *Lazy Data Structure*

In order to optimise the unit propagation, modern solvers do not store the whole current simplified formula regarding the current assignment. Only some literals are kept in order to watch clauses, the most often used structure is the *2-watched literals* [89]. A clause matters during search when: (i) it is going from two non-falsified literals to one because the clause becomes unit; (ii) it is going to one non-falsified literal to zero meaning there is a conflict. Each clause is watched using two literals, with the invariant that if the clause is not satisfied the watched literals are unassigned. When a watched literal becomes falsified, if another unwatched literal is unassigned it becomes watched, otherwise either the clause has become unit, there is a conflict, or the clause is already satisfied. The advantage of such a mechanism is that fewer clauses are visited when a variable is assigned. Moreover when backjumping, unassigned a literal is free because nothing should be done.

3.6 PREPROCESSING/INPROCESSING

In order to optimise the resolution time, the treated formula can be preprocessed before starting the resolution. This is done using a *preprocessing engine*; different types of techniques exist,

we present here the most important ones. This kind of technique can also be used at some point during the solving: this is called *inprocessing*.

3.6.1 *Simplifying Formulas*

The objective can be to simplify the formula by eliminating variables and clauses. A classical technique is to check the *subsumption* of clauses. Let φ be a formula, and $\omega_1, \omega_2 \in \varphi$ be clauses. If $\omega_1 \subset \omega_2$ then ω_2 can be safely deleted from φ. There exist many other rules (*e.g.,* hidden tautology elimination, variable elimination) to simplify the formula implemented in preprocessor engines such as `SatElite` [40], and `coprocessor` [77, 79]. More recently, an inprocessing applying learnt clauses minimisation (LCM) has been proposed by [74]. This minimisation uses unit propagation in order to derive learnt clauses subsuming already learnt clauses. Such a procedure has an important cost and is only applied at certain restarts and on certain clauses.

3.6.2 *Resolution of Particular Sub-Formulas*

A second approach is to detect a sub-formula (*i.e.,* a subset of clauses) that has a particular form: for example, xor-SAT that is solvable in $O(n^3)$ using Gaussian elimination (see Section 2.2.3). If this sub-formula is UNSAT the search completely terminates and the original formula is thus UNSAT, otherwise the formula should be treated as usual. An algorithm to extract xor gates from CNF formulas is presented in [100]. Moreover, this technique can also be used as inprocessing [107].

3.6.3 *Adding Relevant Clauses to the Formula*

Some preprocessing analyses the formula in order to add relevant clauses to speed up the resolution. This is the case with `modprep` [4] that first represents the formula as a graph and then looks for community structures (see Section 7.2). It then launches, for a small amount of time, a solver on sub-parts of the formula. These parts are created by apportioning the different clauses regarding the communities. From these preliminary searches, good clauses can be learnt and added to the initial formula. This is also the case of *static symmetry breaking* [2, 35] that spots equivalent sub-spaces of the search space allowing the solver to cutoff certain of them by adding clauses to the initial formula. These clauses can also be added during the search as a kind of inpreprocessing, we call that *dynamic symmetry breaking*, this is done for example in [85].

3.7 Application on an Example

This section illustrates the SAT resolution process of CDCL as described in Algorithm 4 on the formula $\varphi_1 = \{\omega_1 = \{a, d, e\}, \omega_2 = \{b, d, \neg e\}, \omega_3 = \{c, \neg d\}, \omega_4 = \{b, f\}, \omega_5 = \{\neg f, \neg g\}\}$.

In this example, we consider that decisions are taken in the lexicographic order and that a variable's polarity is always $\bot$. Let l be a literal; it can be undefined, $\top$, $\bot$ or involved in a conflict, respectively noted: l, $\mathbf{l}$, ~~l~~, or $\boxed{l}$. For each step, the clause database, the decision tree, and the implication graph are shown.

Step 1: Branching $\neg a$. At level 0, there are no literals to be propagated. At level 1, a is chosen as the decision variable (according to the lexicographic order), and is assigned to $\bot$. There is nothing to be deduced using the unit propagation.

Clauses database

$\omega_1 = \{$~~a~~$, d, e\}$
$\omega_2 = \{b, d, \neg e\}$
$\omega_3 = \{c, \neg d\}$
$\omega_4 = \{b, f\}$
$\omega_5 = \{\neg f, \neg g\}$

Decision tree

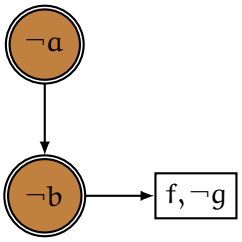
Implication graph

Step 2: Branching $\neg b$. At level 2, b is chosen as the decision variable, and is assigned $\bot$. Unit propagation assigns f to $\top$ using ω_4, and g to $\bot$ using ω_5.

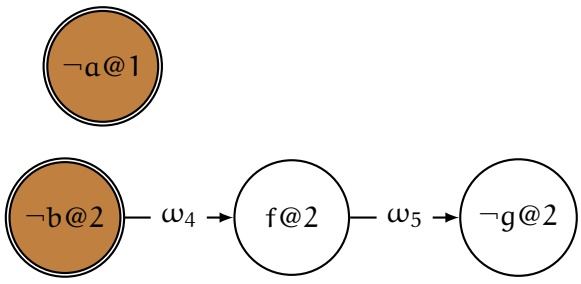
Clauses database

$\omega_1 = \{$~~a~~$, d, e\}$
$\omega_2 = \{$~~b~~$, d, \neg e\}$
$\omega_3 = \{c, \neg d\}$
$\boldsymbol{\omega_4} = \{$~~$b$~~$, \mathbf{f}\}$
$\boldsymbol{\omega_5} = \{$~~$\neg f$~~$, \neg \mathbf{g}\}$

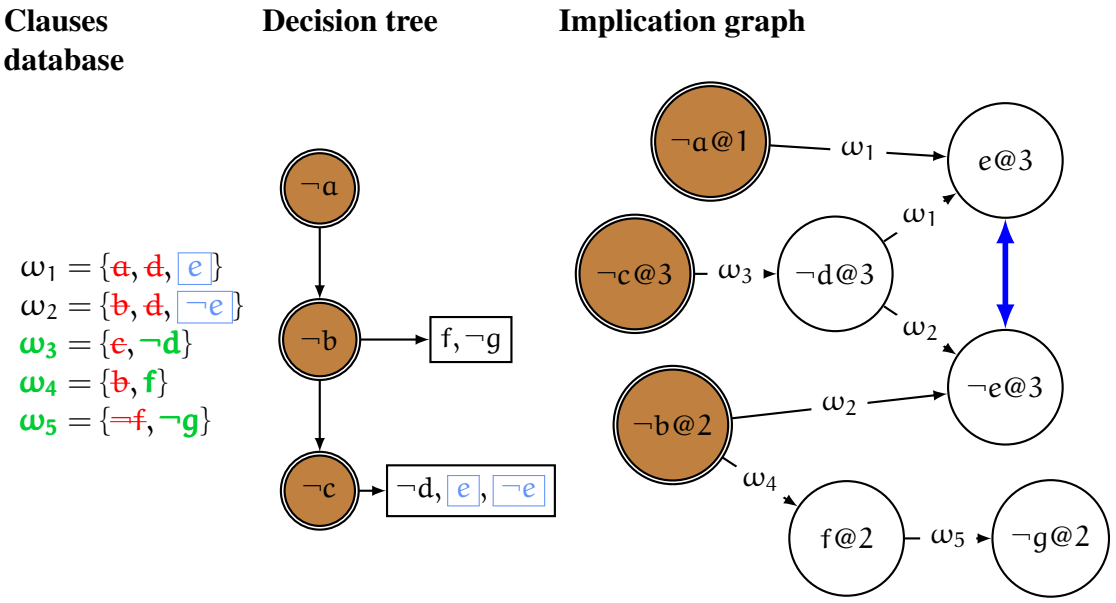
Decision tree



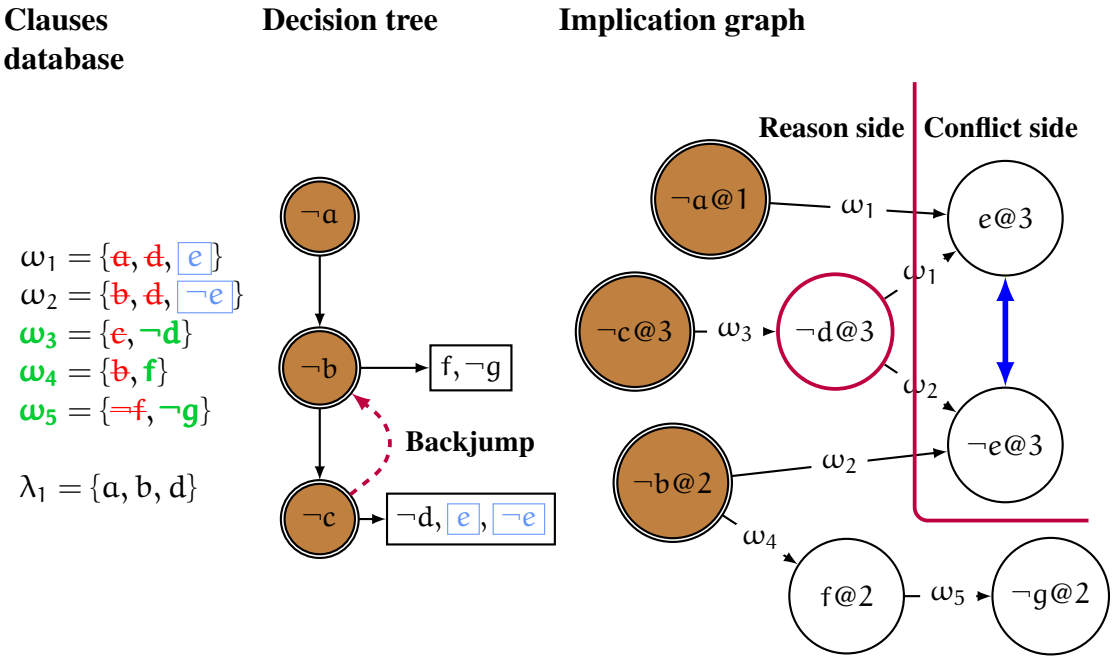
Implication graph



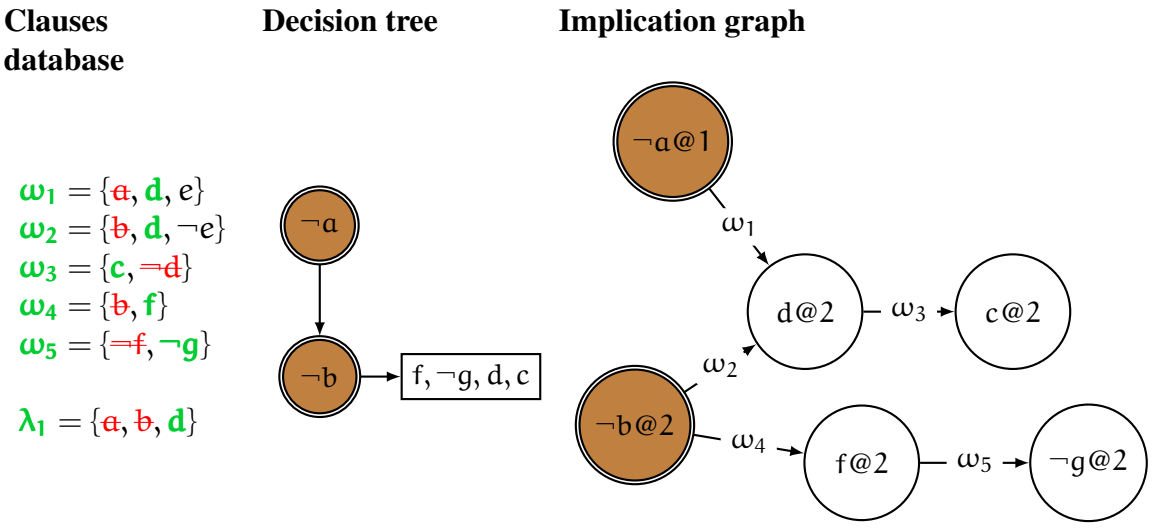
Step 3: Branching $\neg c$. At level 3, c is chosen as the decision variable, and is assigned $\bot$. Unit propagation assigns $\bot$ to d, $\top$ to e, and $\bot$ to e, using respectively ω_3, ω_1, and ω_2. The search reaches a conflict involving e.



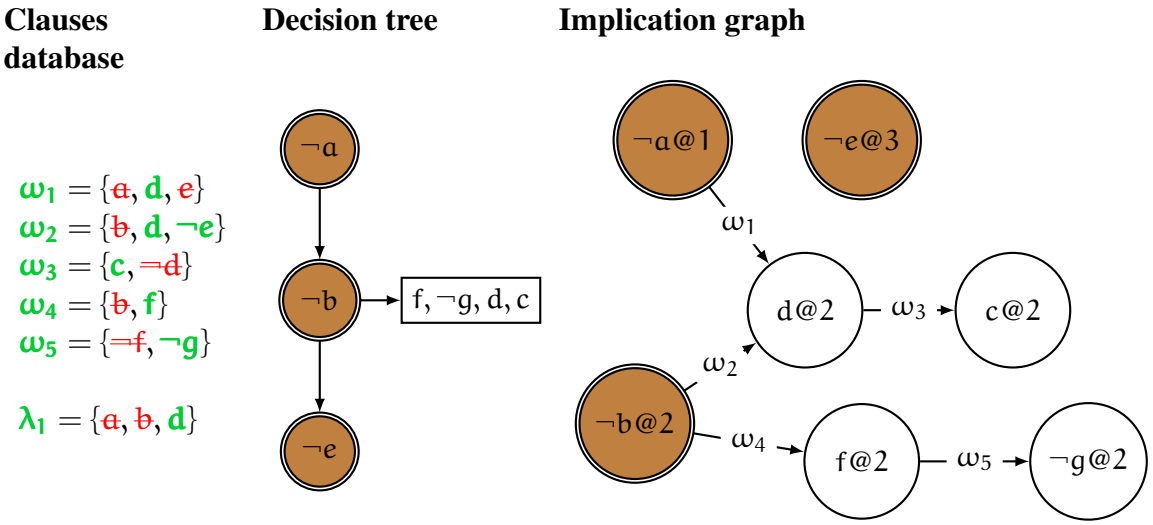
Step 4: Conflict Analysis. Analysis of the conflict shows that $\neg d@3$ is the 1^{st}-UIP, and the cube $[\neg a, \neg b, \neg d]$ is identified as the reason for the conflict. The learnt clause $\lambda_1 = \{a, b, d\}$ is then added to the original formula, and the search backjumps to level 2.



Step 5: Propagation After Conflict. Based on the enriched problem, unit propagation assigns $\top$ to d and c, using respectively λ_1, and $\omega 3$.



Step 6: Branching $\neg e$. At decision level 3, e is chosen as decision variable, and assigned to $\bot$. All the variables have a value, φ_1 is SAT for the assignment $\alpha = \{\neg a, \neg b, c, d, \neg e\}$, and the algorithm terminates returning $\top$. Note that for this assignment a, b, and e are don't care variables.



3.8 Summary and Discussion

In this chapter we have seen how complete a sequential SAT solver works. CDCL algorithm, VSIDS heuristic, and LBD measure are of the main technical inventions of the last 20 years in the area of complete methods in propositional satisfiability solving. However, since 2009,

novelties in terms of heuristics have became increasingly harder to find and this prompted the focus on other research directions. We can mention the following:

(1) understand the underlying phenomena by use of machine learning models. The aim is to derive new unexplored heuristics and optimisations based on mathematical models produced by the machine learning algorithms [70, 72].

(2) Exploit the parallelism paradigms. With the emergence and democratisation of massively parallel machines, the adaptation of SAT solving algorithms and solvers becomes of great interest [112, 49, 50, 59].

(3) Study and exploit problems' structures and properties. For example, the community structures [92, 4] and symmetry properties [2, 35, 85] observed in industrial and handcrafted problems. Solvers can thus specially tuned to handle efficiently such kinds of problems.

(4) Develop decision procedures for higher order logics. The best example here is all the advances around SAT Modulo Theory (SMT), both from the theoretical and practical aspects [20, Chapter 26].

As we have seen in this chapter there are many contributions introducing new sophisticated heuristics and optimisations for sequential SAT solving. The first one is the DPLL algorithm that was proposed in 1962 [32, 31]. These contributions often target a special component embedded in SAT solvers (*e.g.,* decision, polarity, clause management, preprocessing). A SAT solver is a composition of chosen instantiation for these different components. Hence, research and development in the field of complete SAT solving have been clearly simplified by the introduction of the modular solver `MiniSat` [38]. Sixteen years after its first release, many of the available sequential solvers are based on it.

Chapter

4

Parallel SAT Solving

Contents

In the past, processor capacities were increasing with higher frequencies, which allowed sequential SAT solvers to be faster. More recently, due to energy efficiency, ship manufacturers have shifted to multi-core architectures. As a result, new computers have many processing units on a single chip. Moreover, with the massive use of cloud computing, environments such as clusters and grids are more common. All this computing power has been taken into account to improve SAT solving. New strategies have been developed and SAT solving has entered a new era. The goal of this chapter is to highlight and classify the different approaches that have been used to parallelise SAT solving.

We are aware of four main approaches which have been studied to parallelise SAT solvers. First, as the operation where a sequential solver spends the most time is unit propagation, authors of [78, 59] tried to parallelise it. They concluded that this is not a good solution, indeed the different propagations are too dependent on each other, leading to a poor gain according to Amdahl's law. Secondly, some instances are so big that they cannot be contained entirely in the memory of a single computer. A solution, proposed for instance in [50, 102], consists of dividing the formula and allowing the resolution of the initial problem part by part. This solution becomes complicated when all the sub-formulas are proven SAT as they share variables, the original formula is then not necessarily SAT. The third one is divide-and-conquer (D&C) where the search space is dynamically divided and relies on *dynamic load balancing*. The fourth technique is portfolio: it assigns all the workers (*i.e.,* sequential solvers) to the original formula. Portfolio is based on the two concepts of *diversification* and *intensification*. In this chapter we only focus on the last two approaches.

Section 4.1 introduces useful background about parallel environments. Section 4.2 deals with the divide-and-conquer paradigm, and Section 4.3 focuses on the portfolio one. Section 4.4 presents hybrid architectures. In Section 4.5 we introduce the concept of learnt clause sharing. Section 4.6 concludes this chapter and highlights different problems we have identified.

4.1 ABOUT PARALLEL ENVIRONMENTS

This section introduces useful background in order to deal with parallel machines with multi-core architectures, and distributed environments such as grids, clusters, and clouds. This knowledge is crucial in order to design efficient scalable parallel applications. Note that we are not interested in our study in graphical processing units (GPU) [33, 86, 45, 43, 15, 28, 5, 29], nor field-programmable gate array (FPGA) [98, 114, 103], that have been used to parallelise SAT solving, but do not seem to provide yet interesting results.

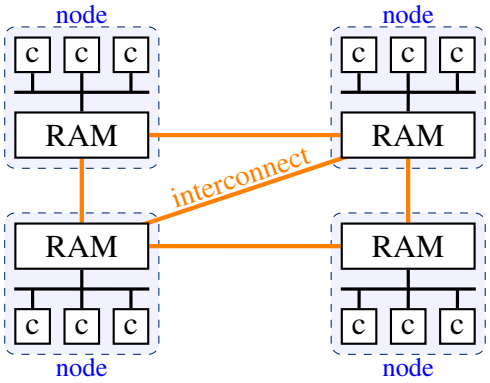


Figure 4.1: A modern NUMA system

4.1.1 *Multi-core Environments*

To understand how new multi-core architectures work, we first introduce some vocabulary. A *core* is a simple processing unit. With *hyper-threading*[1], the operating system considers see two *logical cores* when there is in fact only one *physical core*. A group of cores connected to the same bus is called *socket*. Hyper-threading performances are clearly not equal to performances while doubling the number of cores. Only a gain up to around 20% can be expected when using hyper-threading, and clearly depends on the considered application [82]. As an example, winners of the parallel track of the SAT Competition 2017 and 2018 used only the number of physical cores.

All these cores access data for processing. These accesses are hierarchical and do not have the same latency cost. Each core has its registers, and its own data caches (*i.e.,* L1, L2). Another level of cache (*i.e.,* L3) is shared between the different cores of the socket. Finally cores can access the RAM (Random Access Memory). Since multiple processes can share data, all of these caches should be coherent. This coherence is enforced by the MOESI protocol (see [1, Section 7.3]). A careful implementation, aware of this data accesses hierarchy, can clearly speedup the designed SAT solvers, for instance even in sequential up to 40% according to [76].

Moreover, in massive multi-core machines with tens of cores, the RAM is fragmented in multiple regions. These kinds of architectures are called NUMA (Non-Uniform Memory Access) and are becoming mainstream. Cores are organised in memory regions called nodes (*a.k.a.,* sockets) as shown in Figure 4.1. If a core wants to access data stored in the RAM of another node, its request will pass through *interconnect* links (orange links in Figure 4.1). NUMA architectures introduce new phenomena that programmers should be aware of in order to develop efficient applications. The biggest problems are overloading of a particular node, and contention on interconnect links. This has been pointed out in [30].

[1]This technology is often known under the name hyper-threading, this is the name of the Intel technology, but the generic name of such a technique is simultaneous multi-threading (SMT).

4.1.2 *Distributed Environments*

In order to perform large computations, more and more infrastructures composed of multiple machines such as clusters, grids, and clouds are becoming available. In these kinds of environments different computers are connected via a network.

Cluster. It is a group of collocated machines connected to the same network. Today, this network is often fast (*e.g.,* InfiniBand, Gigabit Ethernet, 10GibE). This type of infrastructure is administrated by a single organisation, therefore it can be used easily. Two processes on two machines of the same cluster can efficiently exchange data using message passing.

Grid. It is a set of clusters, often far from each other. For instance, Grid5000 [24] is a French grid containing clusters sometimes separated by hundreds of kilometers. Therefore, data exchanges become very costly. Moreover, the infrastructure is shared among many users and the computing time is allocated using reservation scheduling policies. The bigger time slot is requested, the longer the waiting time is. Computation can be reserved by an application running on a personal machine through an API.

Cloud. A cloud is a grid in which resources are shared and dynamically allocated to the users according to their needs. This sharing of the resources can be enforced by the use of virtualisation (*e.g.,* containers, hypervisor).

In this section we have described the different kinds of environments that can be used to run parallel SAT solvers. The strategies the solver will rely on should take into consideration the targeted environment. The first difference is the time of information exchanges. These exchanges on a single machine are done by sharing memory and allow a higher debit compared to communications via message passing between different machines. In the case of a parallel SAT solver, this technical difference should be considered to implement an efficient clause sharing (see Section 4.5.3). Moreover, with distributed environments, due to the large number of machines, the probability that a machine fails or becomes unreachable is increased. To deal with these infrastructures, applications such as parallel SAT solvers should be at least crash tolerant (see Section 4.4). Building a crash tolerant distributed SAT solver is more or less difficult depending on the used parallel strategy. For instance, it is easier to design a crash tolerant portfolio than a divide-and-conquer. Indeed in a portfolio all the worker browse the same search space.

4.2 DIVIDE-AND-CONQUER

The divide-and-conquer strategy, one of the two main techniques to parallelise SAT solvers, is based on splitting the search space into subspaces that are submitted to different workers.

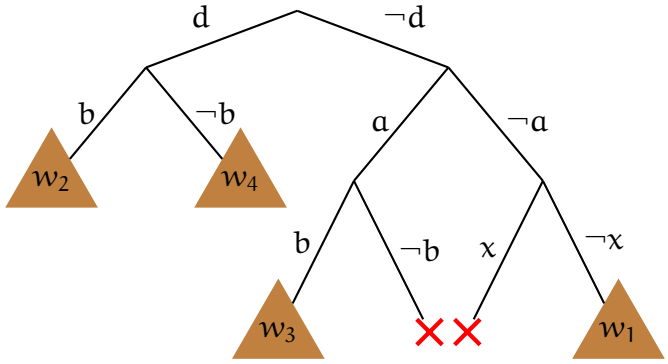


Figure 4.2: Using guiding path to divide the search space

If a subspace is proven SAT then the initial formula is SAT. The formula is UNSAT if all the subspaces are UNSAT. The challenging points of the divide-and-conquer mechanism are: (1) *how to divide the search space*; (2) *finding heuristics to balance their estimated computational costs*; (3) *dynamically balancing jobs between workers*; and (4) *exchanging learnt clauses*.

4.2.1 *Techniques to Divide the Search Space*

To divide the search space, the most often used technique is the *guiding path* [112]. It is worth noting that other partitioning techniques exist, we can cite the *scattering* [55], and the *xor partitioning* [96] approaches.

Guiding Path. It is a conjunction of literals (*i.e.,* cube) that are assumed by the invoked solver (worker). Let φ be a formula, and $x \in \mathcal{V}$ a variable. Thanks to Shannon decomposition, we can rewrite φ as $\varphi = (\varphi \wedge x) \vee (\varphi \wedge \neg x)$. The two guiding paths here are reduced to a single literal: (x) and $(\neg x)$. This principle can be applied recursively on each subspace to create multiple guiding paths.

Figure 4.2 illustrates such an approach where six subspaces have been created from the original formula. They are issued from the following guiding paths: $(d \wedge b)$, $(d \wedge \neg b)$, $(\neg d \wedge a \wedge b)$, $(\neg d \wedge a \wedge \neg b)$, $(\neg d \wedge \neg a \wedge x)$, $(\neg d \wedge \neg a \wedge \neg x)$. The subspaces that have been proven UNSAT, are highlighted with red crosses. The rest of the subspaces are submitted to workers (noted w_i).

Scattering. It works as follows, for a given formula φ and n, a number of partitions (also called *scattering factor*), the construction of the sub-formulas φ_i is given by:

$$\varphi_i = \begin{cases} \varphi \wedge \gamma_1, \text{ if } i = 1 \\ \varphi \wedge \neg\gamma_1 \wedge \ldots \wedge \gamma_{i-1} \wedge \gamma_i, \text{ if } 1 < i < n \\ \varphi \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_n, \text{ if } i = n \end{cases}$$

Where, γ_i is a cube of d_i literals of φ: $\gamma_i = (l_1 \wedge \ldots \wedge l_{d_i})$. The negation of the cube γ_i is the clause $\neg\gamma_i = (l_1 \vee \ldots \vee l_{d_i})$. The number d_i is chosen such that φ is divided in n equal size parts, this is done by minimising $|2^{-d_i} - (n - i + 1)^{-1}|$.

Xor Partitioning. With the use of xor partitioning, for a formula φ and n literals, φ can be partitioned in two sub-formulas: $\varphi_{even} = \varphi \wedge (l_1 \oplus \ldots \oplus l_n \oplus 1)$ and $\varphi_{odd} = \varphi \wedge (l_1 \oplus \ldots \oplus l_n \oplus 0)$. When solving φ_{even} (φ_{odd}), an even (odd) number of the n literals should be assigned to true in order to satisfy the xor constraint. The resulting partitions can be recursively split if more partitions are needed. The advantage of this technique is that if the considered formula is SAT, it is more likely that the models are balanced between the generated subspaces. Note that using this mechanism with $n = 1$ is equivalent to the guiding path technique.

4.2.2 *Choosing Division Variables*

Choosing the best *division variable* is a hard problem, requiring the use of heuristics. A good division heuristic should decrease the overall total solving time[2]. Besides, it should create balanced subspaces w.r.t. their solving time: if some subspaces are too easy to solve this will lead to repeatedly asking for new jobs and redividing the search space (the phenomenon is known as *ping-pong effect* [61]).

Division heuristics can be classified in two categories: *look ahead* and *look back*. Look ahead heuristics rely on the possible future behaviour of the solver. Contrariwise, look back heuristics rely on statistics gathered during the past behaviour of the solver. Let us present what we consider to be the most important ones.

4.2.2.1 *Look Ahead*

In stochastic (incomplete) SAT solving [20, Chapters 5 and 6], look ahead heuristics are used to choose the variable implying the biggest number of unit propagations as a decision variable. When using this heuristic for the division, one tries to create the smallest possible subspaces (*i.e.,* with the least unassigned variables). The main problem of this technique is the generated cost of applying the unit propagation for the exploration of the different candidate variables. Moreover, only one of these call to the unit propagation procedures (the one derived from the selected division variable) will be kept. The `MTSS` [109] solver, and the so-called *cube-and-conquer* paradigm [54] relies on such a heuristic.

4.2.2.2 *Look Back*

This section describes look back heuristics used to select division variables.

[2] Compared to the solving time using a sequential solver

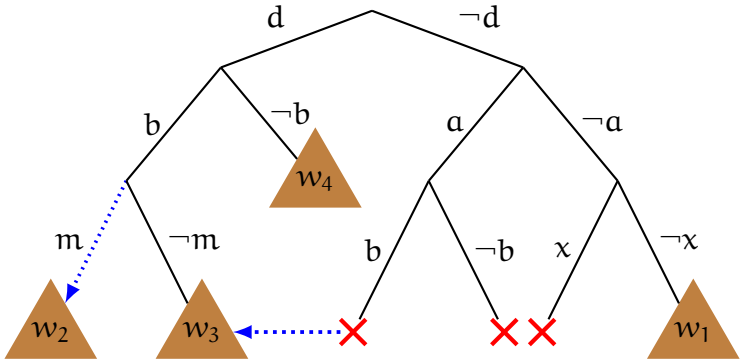


Figure 4.3: Dynamic load balancing through work stealing

VSIDS. Since sequential solvers are based on heuristics to select their decision variables, these can naturally be used to operate the search space division. The idea is to use the variables' VSIDS-based [88] order[3] to decompose the search in subspaces. Actually, when a variable is highly ranked w.r.t. to this order, then it is commonly admitted that it is a good starting point for a separate exploration [55, 83, 9].

Number of Flips. Another explored track is the number of *flips* of the variables [7]. A flip is when a variable is propagated to the reverse of its last propagated value. Hence, ranking the variables according to the number of their flips, and choosing the highest one as a division point helps to generate search subspaces with comparable computational time. This can be used to limit the number of variables on which the look ahead propagation is applied by preselecting a predefined percentage of variables with the highest number of flips.

Propagation Rate. Another look back approach, called propagation rate (PR), tends to produce the same effect as the look ahead heuristic (presented in Section 4.2.2.1) by analyzing gathered statistics. The PR of a variable v is the ratio between the numbers of propagations due to the branching of v divided by the number of time v has been chosen as a decision. The variable with the highest PR is chosen as the division point.

4.2.3 *Dynamic Load Balancing*

Despite all the effort to produce balanced subspaces, it is practically impossible to ensure the same difficulty for each of them. Hence, some workers often become quickly idle, thus requiring a dynamic load balancing mechanism.

Work Stealing. A first solution to achieve dynamic load balancing is to rely on *work stealing*: each time a solver proves its subspace to be UNSAT[4], it asks for a new job. A target

[3] The number of their implications in propagation conflicts.
[4] If the result is SAT the global resolution ends.

worker is chosen to divide its search space (*e.g.,* extends its guiding path). Hence, the target is assigned to one of the new generated subspaces, while the idle solver works on the other. The most common architecture to implement this strategy is based on a master/slave organisation, where slaves are solvers.

When a new division is needed, choosing the best target is a challenging problem. For instance, the `Dolius` solver [7] uses a FIFO order to select targets: the next one is the worker that is working for the longest time on its search space. This strategy guarantees fairness between workers. Moreover the target has a better knowledge of its search space, resulting in a better division when using a look back heuristics.

Let us suppose in the example of Figure 4.2 that worker w_3 proves its subspace to be UNSAT, and asks for a new one. Worker w_2 is chosen to divide and share its subspace. In Figure 4.3, $\mathfrak{m}$ is chosen as division variable and two new guiding paths are created, one for w_2 and one for w_3. Worker w_3 now works on a new subspace and its new guiding path is $(\mathfrak{d}, \mathfrak{b}, \neg\mathfrak{m})$, while the guiding path of w_2 is $(\mathfrak{d}, \mathfrak{b}, \mathfrak{m})$.

Work Queue. Another solution to perform dynamic load balancing is to create more search subspaces (jobs) than available parallel workers (cube-and-conquer [54]). These jobs are then managed via a work queue where workers pick new jobs. To increase the number of available jobs at runtime, a target job is selected to be divided. The strategy implemented in `Treengeling` [18] is to choose the job with the smallest number of variables; this favours SAT instances.

Guiding Tree Structure. The *guiding tree* [109] structure extends the guiding path principle. It has been designed for multi-core platforms (see Section 4.1) and is based on the notions of: *rich worker* and *poor worker*. The rich worker is a classical sequential solver which is helped by poor workers. The current exploration of the rich worker generates a guiding path. The guiding tree is the set of guiding paths that are not part of the search subspace currently explored by the rich worker. Poor workers are autonomous and explore these guiding paths. Their goal is to get information (*e.g.,* preprocessing, splitting) about these subspaces. Indeed they can also prove their current subspace UNSAT or SAT. Information is different depending on the underlying solvers or global strategy. This information is directly stored in the guiding tree structure and will be used by the rich worker when it backtracks.

4.2.4 *Exchanging Learnt Clauses*

Clause sharing in a divide-and-conquer organisation should be restricted depending on the manner the search space division is enforced. Dividing the search space can be subsumed to the definition of constraints on the values of some variables. Technically, there exist two manners to implement such constraints:

(i) constrain the original formula;

(ii) constrain the decision process initialisation of the used solver.

When the search space division is performed using (i), some learnt clauses cannot be shared between workers. This is typically the case of learnt clauses deduced from at least one clause added for space division, otherwise, correctness is not preserved. The simplest solution to preserve correctness is then to disable clause sharing [18]. Another (more complex) approach is to mark the clauses that must not be shared [63]. Clauses added for the division are initially marked. Then, the tag is propagated to each learnt clause that is deduced from at least one already marked clause.

When the search space division is performed using (ii), some decisions are forced. With this technique there is no sharing restrictions for any learnt clauses. This solution is often implemented using the assumption mechanisms, as in `Dolius` [7] and `AmPharoS` [9].

4.3 PORTFOLIO

The portfolio scheme was introduced with the solver `ManySat` [49] and is nowadays dominating the parallel SAT world[5]. In a portfolio all solvers work on the entire formula, the first one to find a solution ends the computation. Two concepts are important to understand this technique: *diversification* and *intensification*.

4.3.1 *Diversification*

The diversification is a variation chosen to vary workers' behaviour. The goal is to visit the search space differently in order to increase the chance to quickly ends the solving. Different strategies to enforce diversification exist that can be used individually or combined.

Parametrization. A simple way to ensure diversification is to use the same underlying solvers initialised with different values for some parameters. These parameters can be random seeds, thresholds, etc.

Heuristic Composition. Since there exists multiple heuristics in the sequential context, one can instantiate solvers with different heuristics for the different components embedded by the sequential engine. Their difference can be on the decision strategy, the restarts, the learning schemes, etc. For instance, in `ManySat` [49] four workers are used, they have differences on their restart strategies, decision heuristics, and learnt clause schemes. It can also be done by using different sequential solvers at the same time as in `HordeSat` [14]. This last strategy has been also used in `pfolioUZK` [111], a simple script solver that won the SAT Challenge 2012, and runs different solvers in parallel depending of the number of available cores.

[5] Regarding the different solvers performing well in last SAT Competitions.

Soft Division. A second strategy to ensure diversification can come from playing with the workers phase. In `HordeSat`, before starting the search each solver receives a special phase. This actually acts as a soft division of the search space. Solvers are invited to visit a certain part of the search space but they can move out of this region during the search.

Intention. Others have proposed to modify the behaviour of solvers based on their *intention* [47]. A similarity in the intention is computed for each pair of solvers using a Hamming distance on their respective phase. If two solvers have similar intentions, one of them will have its phase inverted. In [47], intentions between pairs of solvers are computed every 5000 conflict. Two solvers have a similar intention if the Hamming distance between their phase is less or equals to 0.1.

Block Branching. A third technique to ensure the diversification is *block branching* [105]. Each worker is focused on a particular subset (or block) of variables. For each worker, the decision ranking score (*e.g.,* VSIDS) of variables it is in charge of are periodically bumped. Using, for instance *counter implication restart* (CIR) [104], VSIDS of these variables are vigorously increased every fixed number of restarts. This strategy forces workers to choose decision variables in their own subset. A manner to obtain these subsets is to analyse the relationship of variables using a Union-Find structure as described in [105]. Variables are then merged using binary clauses present in the initial CNF formula. Another idea is to use community structure of the formula represented as a graph [106] (see Section 7.2).

4.3.2 *Intensification*

The concept of intensification has been introduced in [46]: some workers (slaves) re-explore regions of the search space that have been already partially explored by another worker (master), but differently. This *intensifies* the global knowledge on this part of the search space. From the initial exploration of particular search space, several information is known: the set of decision literals $\mathcal{D}$ (*i.e.,* taken decisions and their polarity); and Λ, the set of clauses learnt during the search. The set Λ is added to the slave to warrant that it will not do the same work as the master. Moreover, this information is used to construct a set of possible decision literals (*i.e.,* decision variables plus their polarity) for the slave in charge of the intensification. Authors of [46] proposed three approaches. In the first one, the slave will take exactly the same decision literals (*i.e.,* $\mathcal{D}$). In the second technique, they use as decisions literals the asserting set (*i.e.,* the negation of 1^{st}-UIP of IG that have lead to learnt clauses in Λ). The third solution uses as decision literals the conflict set (*i.e.,* literals included between the 1^{st}-UIP and the conflict variable of IG that has led to learn a clause in Λ). The third technique seems to be the most efficient one according to the conducted experiments. When the master restarts, the slaves also restart and enter a new intensification phase.

Diversification and intensification are two orthogonal axes. If workers involved in the diversification process are called masters and the ones that are in charge of intensification are viewed as slaves, authors of [46] give us the best configuration: each master should have one slave.

4.4 Hybrid Strategies

As presented above, portfolio (see Section 4.3), and divide-and-conquer (see Section 4.2), are the two main approaches explored to parallelise SAT solving.

The portfolio scheme is very simple to implement, and uses the principle of diversification to increase the probability of solving the problem faster. However, since worker search regions can overlap, the theoretical resulting speedup is not as good as the one of the divide-and-conquer approach [57]. While giving a faster theoretical speedup, the divide-and-conquer approach suffers from two challenging issues we have already mentioned: dividing the search space and load balancing.

One idea to increase the effectiveness of parallel SAT solving, and benefit from the two worlds, is to use simultaneously the two strategies. These are called *hybrid approaches*.

4.4.1 *Composition of Strategies*

A basic manner to mix the two approaches is to compose them. This composition results in two possible strategies: portfolio of divide-and-conquer, and divide-and-conquer of portfolio. The former has been introduced by `c-sat` [93], and targets clusters (see Section 4.1). Each machine is in charge of the whole resolution (*i.e.,* portfolio), on each machine a divide-and-conquer approach is used to create jobs for the different processors. Here machine crashes are not important since every machine works on the whole formula. The second approach is implemented in the solver `AmPharoS` [9], where a divide-and-conquer is used, but workers are free to choose their search space. Solvers stop after k conflicts ($k = 10000$ in the implementation) and can then solve another search space. It is then possible to have multiple workers on the same search space.

4.4.2 *Partition Tree Approach*

Another approach called *partition tree* [55] recursively divides the search space, but every created subspace is treated, even if it has been partitioned. In the partition-tree, each node represents a job, and some are thus redundant. This technique has been proposed for grid environments. Crashes have then less impact. This technique also performs well on multi-core architectures as pointed out in [58].

4.4.3 *Approaches using Transition Heuristics*

In order to use portfolio or divide-and-conquer when it is the more appropriate, *transition heuristics* are introduced to decide when the search should switch from portfolio to divide-and-conquer and vice-versa.

In [22], workers are organised in a divide-and-conquer, when a certain subspace is not resolved after k conflicts (k is initially equal to the number of variables of the formula, and is doubled every time a transition is issued), the organisation for its resolution switches to portfolio (*i.e.,* the corresponding job is duplicated in the work-queue). To avoid the organisation to derive in a simple portfolio, the transition heuristic is used only if the subspace has been divided at least once.

In [99], the authors proposed a peer to peer approach. They use a divide-and-conquer strategy. Certain subspaces can be resolved using a portfolio if they are too hard to solve. The difficulty of subspaces is determined using the *task aggregated split quality* (TASQ), that is a weighted average of the *split quality* over time of the subspaces derived from the analysed subspace. The split quality is based on the time already spent trying to solve a particular subspace. TASQ indicates if a subspace should continue to be solved using a divide-and-conquer approach or a portfolio one.

The resolution process of `SAT4J//` [83] is composed of three phases. The first phase uses a portfolio till encountering a certain number of conflicts (in order to collect information). This first phase is also present in the solver `AmPharoS`. The second phase is a divide-and-conquer. There are two ways to decide when to stop this second phase: when a search space has not been solved after k conflicts and when more than half of jobs have been derived from it; or it has not yet been solved after $z >> k$ conflicts. If one of these two situations is reached, the search switches to a last portfolio phase.

In the `AmPharoS` [9] solver, a worker can choose the subspace it will work on. The worker chooses in priority jobs that are not currently treated by another worker. The fewer jobs there are the more the solver tends to be in a portfolio scheme. To manage the extension two criteria are used: the difficulty of the subspace, and the redundancy of shared clauses. The difficulty of the search space is computed by incrementing a counter β each time a worker cannot solve this job after k conflicts. To analyse shared clauses, the *redundancy shared clauses measure* (rscm) is proposed. It is a ratio between the number of clauses exchanged and the number of clauses kept after subsumption for a certain sliding window. A low rscm indicates that shared clauses are not redundant and the extension should be reduced. A high value indicates that redundant clauses are shared and extension should be performed. An extension factor f_e is computed using the rscm as follows: $f_{e_t} = \frac{1000}{\text{rscm}_t}$. A subspace is divided if $\beta \times f_e$ is greater than or equal to the number of jobs. Note that f_e as a fixed minimum value of 10.

4.5 Learnt Clause Exchanges

The most important information that can be shared is the clauses learnt by each worker during its solving. With the two classical paradigms, divide-and-conquer, and portfolio, but also with the use of hybrid approaches, learnt clause exchanges are possible. In practice not all learnt clauses can be shared between all workers, due to hardware limitations (see Section 4.1), but also to the slowdown impact on the unit propagation algorithm. Hence, the main questions are: *which clauses should be shared? And between which workers?* To answer the first of these questions some clauses should be filtered. There are three moments when filtering can be implemented: when clauses are exported (Section 4.5.1), when clauses are in transit during the sharing phase (Section 4.5.3), and finally when clauses are effectively imported (Section 4.5.2). A simple answer to the second question, adopted in almost all parallel SAT solvers, is to share clauses between all workers. However, a finer (but more complex) solution is to let each worker choose its emitters (Section 4.5.4).

4.5.1 *Export Clauses*

In the sequential context the notion of good clauses is already characterised by some measures (*i.e.,* activity, size, LBD [11]), then one can reuse these measures in the parallel context. In many solvers, only clauses under a certain fixed threshold, for these measures, are shared. For instance, in `ManySat` [49], only clauses up to size 8 are shared.

As these thresholds are heuristics, some have proposed to adapt them during the search. This allows fine control of the flow of learnt clauses during the solving time. Based on the *additive increase, multiplicative decrease* (AIMD) algorithm used in TCP to avoid congestion, authors of [48] proposed to adjust dynamically the size of exchanged clauses between pairs of workers. In `HordeSat` [14] sharing is limited in each round to a certain number of literals (*i.e.,* sum of clauses' size). If too few clauses are shared, the threshold of the LBD/size limit is increased.

4.5.2 *Import Clauses*

First, we recall that in a classical CDCL solver, clauses are stored in a lazy way, using the 2-watched schema (see Section 3.5.4).

LBD and activity are measures that are local to a worker. A clause that is good, based on these measures, can be irrelevant in the context of another worker. When clauses are received by a worker, it can check whether clauses will be useful for its current exploration. In order to compute the relevance of the received clauses, measures such as *progress saving based quality measure* (PSM) [8] can be used. The PSM of a clause is the cardinality of the intersection between the clause (seen as a set of literals) and the phase of the worker. A high value is worse since it means that the clauses will be too quickly satisfied by the current exploration. Moreover, in [8] the authors propose not to trash clauses with high PSM, instead these clauses are put in a *freeze* (or *standby*) mode and can be incorporated latter in the solvers database

if their PSM has decreased. If it is not the case they can be trashed definitively. A clause in *standby* mode is not watched by the solver, and is then not used neither for unit propagation nor for conflict detection.

In `Syrup` [13], when a clause is imported, it is added to a *1-watched* structure, where only one of its literals is observed. A structure that still allows the detection of conflict but clauses cannot be used for unit propagation. When a 1-watched clause triggers a conflict, it is promoted to the classical 2-watched mechanism. The idea here is not to overload the unit propagation with imported clauses and only incorporate new clauses that are relevant (*i.e.,* that have already provoked a conflict). This database using 1-watched structure is often called *purgatory*.

Clauses can be managed using a promoting mechanism with different states. For instance, in the solver `AmPharoS` [9], there are three states: `standby`, `purgatory`, and `learnt` (*i.e.,* classical learnt database of the sequential solver). When a clause is received, it is placed in the `standby` state: the clause is not yet attached to the sequential solver. Clauses in the `purgatory` or in the `learnt` states are attached to the solver. Periodically a certain number of clauses are promoted from `standby` to `purgatory`, this number depends of the rscm measure (presented in Section 4.4.3). A clause is promoted from `purgatory` to the `learnt` state when it has been used at least once in a conflict analysis. If a clause is not promoted after multiple reviews (14 in `AmPharoS`), it is consequently deleted. This kind of strategy avoid the unit propagation mechanism of underlying solvers to be flooded by the number of learnt clauses.

4.5.3 *Sharing Phase*

Between the export of a clause by a solver and its import by another solver, there is a sharing phase. The implementation of this phase clearly depends on the environment the solver is executed on. For a solver targeting a single multi-core machine, sharing is achieved using buffers and shared memory [13]. For distributed environments, clause exchanges are done through message passing, and a dedicated thread is needed per machine to manage send/receive clauses from/to other machines [14, 10, 39]. During this sharing phase, a filter can be applied on the clauses in transit. The advantage of such a moment is the most omniscient view of clauses that allowed the use or more complicated heuristics. This filter can remove subsumed clauses [56, 9], remove already seen clauses using bloom filter [14], or operate minimisation [110]. This phase requires a particular focus when implementing a solver and is the most important synchronisation point in a parallel SAT solver.

Learnt clauses exchanges can speed up the solving but are not necessary for the correctness of its answer. Hence, exchanges can be done in a best effort way, for instance in `HordeSat` if the solver exporting clauses need to wait for synchronisation, the clauses is not exported.

4.5.4 *Selecting Emitters*

In almost all parallel SAT solvers, clauses are exchanged between all workers. Another idea, introduced in [64], is to let each worker select the ones that are allowed to send it clauses. This problem can be seen as an optimisation problem and reformule as a Multi-Armed Bandit (MAB) problem. For a given worker, clauses sent by each worker are evaluated using a gain function (*e.g.,* weighted average of received clause activity). The used algorithm is UCB2. The workers sending the most relevant clauses are then invited to continue. If a worker has a bad score, it is then removed from the emitters and another one takes its place. In [64], the number of emitters is fixed to half of the total number of workers.

4.6 Summary and Discussion

This chapter has presented a large overview (we hope it is the most exhaustive possible) of techniques used in the literature to parallelise SAT solving. There are many contributions in this domain, but there are still perspectives to improve parallel SAT solving efficiency.

Following the common intuition, supported by the partial, yet very interesting, analytic study of [57], `D&C` techniques seem to be the natural way to parallelise SAT solving. However, the outcomes of the different, especially the last, editions of the parallel track of the annual SAT contest [6]show that the best state-of-the-art parallel SAT solvers are portfolios. The retained explanation for these results is mainly attributed to the easiness of implementing a portfolio strategy. Contrariwise, `D&C` strategies need hard to implement mechanisms. So far, all efforts to solve these issues are far from being conclusive, and the approaches that try to compose the two classes did not achieve any huge improvements. We believe though that this not the whole story and we still lack some key points in our overall understanding and treatment of the underlying phenomena, both theoretically and practically.

In parallel SAT solving, sharing knowledge between the solvers is crucial. Actually, without this sharing, the collaboration between the parallelisation actors cannot be optimal. Usually, this sharing is implemented as clause exchanges. Each time a solver learn a clause, it can share it with other solvers to prevent them making the same mistake. In this context, three questions are central: (i) what knowledge to share? (ii) With whom to share? And (iii) how to share? Here also, the sharing is fare from being ideal, both theoretically (question (i) and (ii)), and practically (question (iii)). Measures, like LCM, LBD or AIMD, are good starting points to answer questions (i) and (ii), but still we believe that the community should spend some effort defining better quality measures in order to leverage the benefits of knowledge sharing. Besides, a bad implementation of the sharing mechanisms (question (iii)) could collapse the whole performances of the solver.

So far, the evaluation and the comparison of parallelisation approaches is done using different tools: the authors of an approach develop a solver that implements and compares it to another

[6] https://www.satcompetition.org

tool (usually written in another language, with other libraries, and using different programming skills, etc). We believe that such a comparison is biased and does not evaluate fairly the studied approaches. It is then difficult to conclude on the (in)effectiveness of a technique with respect to another one and this may lead to premature abortion of potential good ideas.

As a summary, we can then say that there is still a lot of work to be done at all levels of the subject:

(C1) improving parallelisation strategies. A first challenge would be to fill the gap between `D&C` and portfolio strategies.

(C2) Improving knowledge sharing strategies both theoretically and technically.

(C3) Improving the tooling support. One must be able to integrate a large majority of parallelisation strategies and knowledge sharing, in the same framework. This must be done while preserving easiness, efficiency and modularity.

Chapter

5

Framework for Parallel SAT Solving

Contents

There exist numerous strategies to parallelise SAT solving (see Chapter 4), their performances heavily relying on their implementation. The most difficult issues deal with concurrent programming. Languages and libraries provide abstractions to deal with these difficulties, and according to these abstractions developers have more or less control on features such as memory or threads management. This will affect directly the performance of the produced solver. Therefore, it is difficult to compare the strategies without introducing technological bias. When we want to implement a new strategy for a special component of a parallel SAT solver (*e.g.,* sharing) a complete solver is required. Therefore, there are two possibilities: either integrating this new strategy on top of an existing solver, or developing a new solver from scratch. Clearly the first solution is often considered, but it is then hard to choose a solver among the numerous present in the literature. Moreover, a complete implementation usually offers the possibility to modify a particular component. It is complicated to test our new implementation with multiple combinations of components varying over different axes.

One solution to overcome these problems is to use a modular but efficient framework. There exists modular and efficient framework to parallelise problems with search space exploration. For instance, `bobpp` is a framework library allowing easy implementation of solvers on specific problems using sequential and parallel search algorithms: such as branch and bound, branch and price/cut, divide-and-conquer [44]. The problem of this framework is that it is not designed to massively exchange information (*i.e.,* clauses) between the different workers.

Even if some parallel SAT solvers offers have shown the way by introducing a bit of modularity (*e.g.,* `MTSS` [34], `HordeSat` [14], `Dolius` [7], `AmPharoS` [9]) it is often limited to certain components. For instance, `MTSS` and `HordeSat` allow modularity for sequential solvers and sharing but not for the parallelisation strategy. Concurrent SAT solving still need a modular and efficient framework.

In this chapter we present PArallel INstantiabLE Sat Solver (`Painless`) [66], a framework that aims to be a solution to these problems. It is a generic, modular, and efficient framework, developed in `C++`, allowing an easy implementation of concurrent parallel SAT solvers. Taking black-boxed sequential solvers as input, it minimises the effort to encode new parallelisation and sharing strategies, thus enabling the implementation of complete SAT solvers at a reduced cost.

The rest of this chapter is organised as follows: Section 5.1 presents the architecture of `Painless`. Section 5.2 explains how to use `Painless` for building solvers mimicking state-of-the-art ones. Section 5.3 provides evidence of the modularity and easiness of the proposed solution. Section 5.4 highlights the effectiveness of the different solvers implemented with `Painless`. Section 5.5 presents our solver participant of the SAT competition 2017 and 2018, and Section 5.6 concludes this chapter.

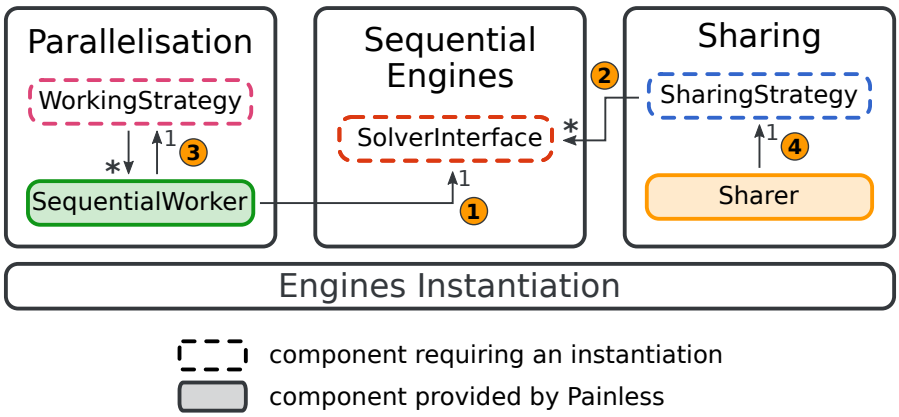


Figure 5.1: General architecture of `Painless`

5.1 Architecture of the Framework

As pointed out by the study of the state-of-the-art in Chapter 4, a typical parallel SAT solver relies mainly on three core concepts: sequential engine(s), parallelisation, and sharing. These form the core of the `Painless` architecture (see Figure 5.1): the sequential engine is handled by the `SolverInterface` component. The parallelisation is implemented by the `WorkingStrategy` and `SequentialWorker` components. Components `SharingStrategy` and `Sharer` are in charge of the sharing. The rest of this section focusses individually on each of these three concept components.

5.1.1 *Sequential Engine*

To manipulate different sequential engines in a transparent way, solvers should be wrapped by a unique common API. `SolverInterface` is an adapter for the basic functions expected from a sequential solver, it is divided in two subgroups: *solving* and *clauses management* (respectively represented by arrows ❶ and ❷ in Figure 5.1). Subgroup ❶ provides methods that interact with the solving process of the underling solver. The most important methods of this interface are:

- `void loadFormula(string filename)`: loads the formula contains in the file named `filename` into the solver;
- `SatResult solve(int[] cube)`: tries to solve the formula, with the given cube (that can be empty in case of portfolio). This method returns SAT, UNSAT, or UNKNOWN.
- `void setInterrupt()`: stops the current search that has been initiated using the `solve` method;

- `void setPhase(int var, bool value)`: set the phase of variable `var` to `value`;
- `void bumpVariableActivity(int var, int factor)`: bumps `factor` times the activity of variable `var`;
- `void diversify()`: adjusts the internal parameters of the solver to diversify its behaviour.

Subgroup ② provides methods to add/fetch (learnt) clauses to/from the solver:

- `void addClause(Clause clause)`: adds a permanent clause to the solver.
- `void addLearntClause(Clause clause)`: adds a learnt clause to the solver.
- `Clause getLearntClause()`: gets the oldest produced learnt clause from the solver.

The interface also provides methods to manipulate sets of clauses (*e.g.,* `void addClauses(Clause [] clauses)`). The clauses produced or to be consumed by the solvers are stored in local *lock-free* queues (based on an algorithm of [87]).

Technically, to integrate a new sequential solver in `Painless`, one needs to create a new class inheriting from `SolverInterface` and implement the required methods (*i.e.,* wrapping the methods of the API offered by the underlying solver).

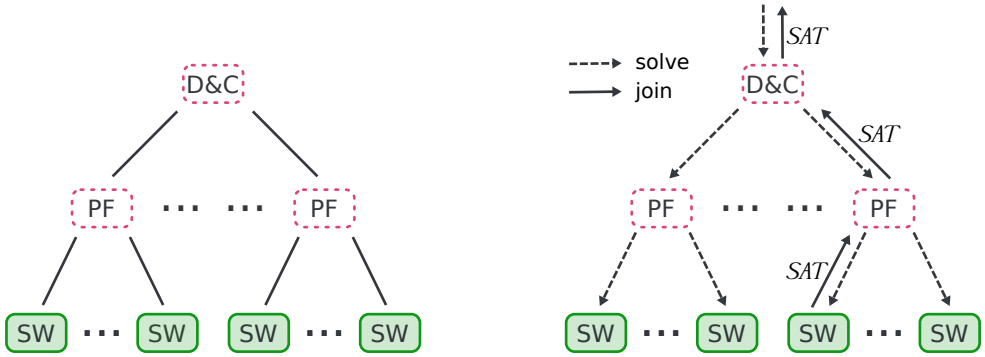
Provided Implementations

- `LingelingAdapter`: an adapter of the `Lingeling` [17] solver;
- `GlucoseAdapter`: an adapter for the `Glucose` [11] solver;
- `MinisatAdapter`: an adapter for the `MiniSat` [38] solver;
- `MapleAdapter`: an adapter for the `MapleCOMSPS` [71] solver.

5.1.2 *Parallelisation*

Basic parallelisation strategies, such as those introduced in Chapter 4 (*e.g.,* portfolio, divide-and-conquer), must be easily implemented. We also aim at creating new strategies and mixing them.

A tree-structured (of arbitrary depth) composition mechanism enables us to mix strategies: internal nodes represent parallelisation strategies, and leaves represent solvers. For instance, in Figure 5.2a a divide-and-conquer of portfolios is represented by a tree of depth 3: the root



(a) Tree structure **(b)** Workflow

Figure 5.2: Example of a composed parallelisation strategy

corresponds to the divide-and-conquer (`D&C`) having children representing the portfolios (`PF`) acting on several sequential workers (`SW`), the leaves of the tree.

`Painless` implements nodes using the `WorkingStrategy` class, and leaves with the `SequentialWorker` class. This last is a subclass of `WorkingStrategy` that integrates an execution flow (a thread) operating its associated sequential solver and is provided by the framework.

The overall solving workflow within this tree is user defined and guaranteed by the two main methods of the `WorkingStrategy` (arrows ❸ in Figure 5.1):

- `void solve(int[] cube)`: according to the strategy implemented, this method manages the organisation of the search by giving orders to the children strategies.
- `void join(SatResult res, int[] model)`: used to notify the parent strategy of the solving end. If the result is `SAT`, `model` will contain an assignment that satisfies the formula treated by this node.

It is worth noting that the workflow must start by a call to the root's `solve` method and eventually ends with a call to the root's `join` method. The propagation of solving orders from a parent to one of its child nodes is done by a call to the `solve` method of this last. The results are propagated back from a child to its parent by a call to the `join` method of this last. The solving cannot be effective without a call to the leaves' `solve` methods.

Back to the example of Figure 5.2a, consider the execution represented in Figure 5.2b. The solving order starts by a call to the root's (`D&C` node) `solve` method. It is relayed through the tree structure to the leaves (`SW` nodes). Here, once its problem is found `SAT` by one of the `SW`, it propagates back the result to its `PF` node parent via a call to the `join` method. According to the strategy of the `PF`, the `D&C`'s `join` method is called, the global solving ends, and the formula is reported to be `SAT`.

Hence, to develop its own parallelisation strategy, the user should create or reuse one or more subclass of `WorkingStrategy` and build the corresponding tree structure.

Provided Implementations

- `Portfolio`: a portfolio strategy (see Section 4.3);
- `CubeAndConquer`: a cube-and-conquer strategy inspired by the `Treengeling` [17] solver;
- `DivideAndConquer`: a generic and modular divide-and-conquer strategy presented in Chapter 6.

5.1.3 *Sharing*

In parallel SAT solving, we must pay particular attention to the exchange of learnt clauses (see Section 4.5). Indeed, beside the theoretical aspects, a bad implementation of the sharing can dramatically impact the efficiency of the solver (*e.g.,* improper use of locks, synchronisation problems). We now present how sharing is organised in `Painless`.

In `Painless` sharing is decentralised and done through the use of lock-free queues [87]. Thus an efficient sharing mechanism is provided to users whom only need to focus on the exchange phase.

When a solver learns a clause, it can share it according to a filtering policy such as the size or the LBD of the clause. To do so, it puts the clause in a special buffer (`buff_exp` in Figure 5.3). The sharing of the learnt clauses is realised by dedicated thread(s): `Sharer(s)`. Each one is in charge of a set of producers and consumers (these are references to `SolverInterface`). Its behaviour reduces to a loop of sleeping and exchange phases. This last is done by calling the interface of `SharingStrategy` class (arrow ❹ in Figure 5.1). The main method of this class is the following:

- `void doSharing(SolverInterface[] producers,`

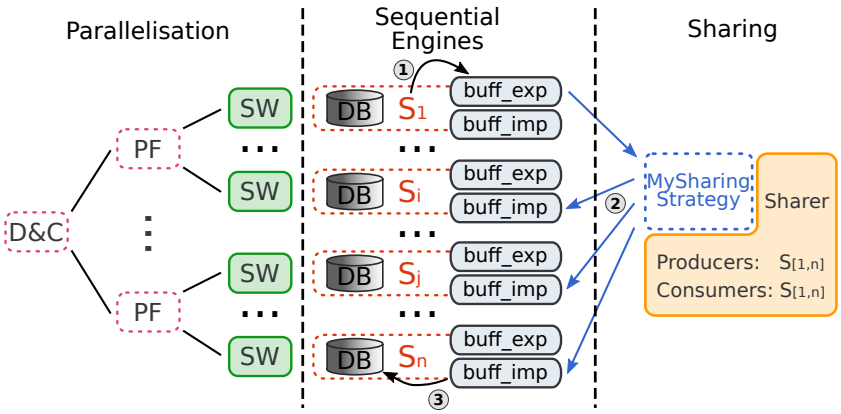


Figure 5.3: Sharing mechanism implemented in `Painless`

`SolverInterface[] consumers)`: according to the underlying strategy, this method gets clauses from the producers and adds them to the consumers.

In the example of Figure 5.3, the `Sharer` uses a given strategy (here named `MySharingStrategy`), and all the solvers (S_i) are producers and consumers. The use of dedicated workflows (*i.e.,* threads) allows CPU greedy strategies to be run on a dedicated core, thus not interfering with the solving workers. Moreover, sharing phases can be done manipulating groups of clauses, allowing the use of more sophisticated heuristics such as the one introduced in Section 4.5.3. Finally, during its search a solver can get clauses from its importing buffer (`buff_imp` in Figure 5.3) to integrate them in its local database. In practice, import is often done when the solver get back to level 0 (*e.g.,* when restarting).

To define a particular sharing strategy the user only needs to provide a subclass of `SharingStrategy`. With our mechanism it is possible to have several groups of sharing each one manage by a `Sharer`. Moreover, solvers can be dynamically added/deleted from/to the producer and/or the customer sets of a `Sharer` allowing, for instance, MAB sharing (see Section 4.5.3).

Provided Implementations

- `SimpleSharing`: a sharing strategy exporting all the received clauses from the producers to consumers;
- `HordesatSharing`: a sharing strategy that increases the flow of learnt clauses if not enough clauses are received from one of the producers [14].

5.1.4 *Engine Instantiation*

To create a particular instance of `Painless`, the user has to adapt the main function presented in Algorithm 6. The role of this function is to instantiate and bind all the components correctly. These actions are simplified by the use of parameters.

First, the concrete solver classes (inheriting from `SolverInterface`) are instantiated (line 2). Then the `WorkingStrategy` (including `SequentialWorker`) tree is implemented (line 3). This operation links `SequentialWorker` to their `SolverInterface`. Finally, the `Sharer(s)` and their concrete `SharingStrategy(s)` are created; the producers and consumers sets are initialised (line 4).

The solving starts by the call to the `solve` method of the root `WorkingStrategy` tree. The main thread will execute a loop, where it sleeps for an amount of time, and then checks if either the timeout or memout has been reached, or the solving ended (lines 6-7). It prints the final result (line 8), plus the model in case of a SAT instance (lines 9-10).

```
1  function painless-main(args: program arguments)
2      solvers ← create SolverInterface
3      root ← create WorkingStrategy tree (solvers)
4      sharers ← create SharingStrategy and Sharer (solvers)
5      root.solve()
6      while timeout or stop do
7          sleep(...)
8      print(root.getResult())
9      if root.getResult() = SAT then
10         print(root.getModel())
```

Algorithm 6: The main function of `Painless`

5.2 IMPLEMENTING EXISTING STRATEGIES

To validate the generic aspect of our approach, we selected three efficient state-of-the-art parallel SAT solvers: `Syrup` [13], `Treengeling` [17], and `HordeSat` [14]. For each selected solver, we implemented a solver that mimics the original one using `Painless`.

5.2.1 *Solver "à la `Syrup`"*

The `Syrup` [13] solver is the winner of the parallel track of the SAT Race 2015 and SAT Competition 2017. It is a portfolio based on the sequential solver `Glucose` [11]. The sharing strategy exchanges all the exported clauses between all the workers. A clause is exported directly when learnt if it is a binary or a glue clause (*i.e.,* LBD = 2). Otherwise, it is exported if the clause is used in conflict analysis or unit propagation, *i.e.,* when it is touched for a second time after being learnt. When a clause is imported it is added in the *purgatory*, and managed through 1-watched (see Section 4.5). Besides, the workers have customised settings in order to diversify their search.

Hence, implementing a solver "à la `Syrup`", namely `P-Syrup`[1], required the following components: `GlucoseAdapter` an adapter to use the `Glucose` solver; `Portfolio` a simple `WorkingStrategy` that implements a portfolio strategy; `SimpleSharing` a `SharingStrategy` that exchanges all the exported clauses from the producers to all the consumers with no filtering. Note that mechanisms to export (*i.e.,* double touched) and managed imported (*i.e.,* purgatory) clauses are embedded in the core solver `Glucose` and do not require any implementation.

[1] For the rest of this manuscript all solvers instantiated with `Painless` will be noted: `P-[\w-]+`.

5.2.2 *Solver "à la Treengeling"*

The Treengeling solver is the winner of the parallel track of the SAT Competition 2016. It is based on the sequential engine Lingeling [17]. Its parallelisation strategy is a variant of divide-and-conquer called cube-and-conquer (see Section 4.2). In Treengeling, the solving is organised in rounds. Some workers search for a given number of conflicts. When the limit is reached, some are selected to split their sub-spaces using a look ahead heuristic. Moreover, beside there is a special sequential solver solving the whole formula. The sharing is restricted to the exchange of unit clauses from the special worker to the ones of the cube-and-conquer.

To implement a solver "à la Treengeling", namely P-Treengeling, we need the following components: LingelingAdapter, an adapter of the sequential solver Lingeling; CubeAndConquer a WorkingStrategy, that implements a cube-and-conquer; SimpleSharing already used to define for the P-Syrup like solver. In this case, the underlying sequential solvers are parametrised to export only unit clauses, and only the special worker is a producer.

For the CubeAndConquer, we choose to manage rounds based on time because it allows, once one worker has encountered an UNSAT situation, to restart the worker with another guiding path. In the original implementation, rounds are managed using numbers of conflicts, this makes the reuse of idle CPU much harder.

5.2.3 *Solver "à la HordeSat"*

The HordeSat [14] solver is a portfolio-based solver with a modular design. HordeSat uses as sequential engine either MiniSat [38] or Lingeling. It is a portfolio where the sharing is realised by controlling the flow of exported clauses: every one second, 1500 literals (*i.e.,* sum of the size of the clauses) are exported from each sequential engine to the others. The selection of clauses is done by considering clauses in increasing order of their size. Moreover, we have used the LingelingAdapter solver and the native diversification of Plingeling [17] (a portfolio solver of Lingeling) combined with the sparse random diversification (presented as the best combination in [14]).

The solver "à la HordeSat", namely P-HordeSat, required the following components: LingelingAdapter and Portfolio that have been implemented earlier; Hordesat-Sharing, a SharingStrategy that implements the HordeSat sharing strategy.

5.3 Modularity Evaluation

This section tries to highlight the modularity and easiness of Painless. It is difficult to quantify the easiness of developing different solvers. A manner we have found to quantify this is the number of lines of code (LoC) required to build the different pieces of software.

Secondly, to show the modularity of `Painless`, we used the already developed components to instantiate two new original solvers that combine existing strategies.

5.3.1 *Factorisation*

Figure 5.4 summaries the effort (given as the required LoC) to implement the three above-mentioned solvers: `P-Syrup`, `P-Treengeling`, and `P-HordeSat`.

The code provided by `Painless` represents 1279 LoC. This code provides many useful facilities: the definition of the different APIs, `SequentialWorker`, `Sharer`, lock-free queues for sharing, clause structure, debug functions, and main plus parameterisation.

Without factorisation the different instances of solvers implemented using `Painless` require the following number of Loc:

- the implementation of `P-Syrup` required 352 LoC for the `GlucoseAdapter` (GA), 93 LoC for the `Portfolio` (PF), and 45 LoC for the `SimpleSharing` (SS). In total `P-Syrup` required a total of 1789 LoC.
- The implementation of `P-Treengeling` needed 375 LoC for the `LingelingAdapter` (LA), 245 LoC for `CubeAndConquer` (C&C), and 45 LoC for the `SimpleSharing`. In total `P-Treengeling` required a total 1944 LoC.

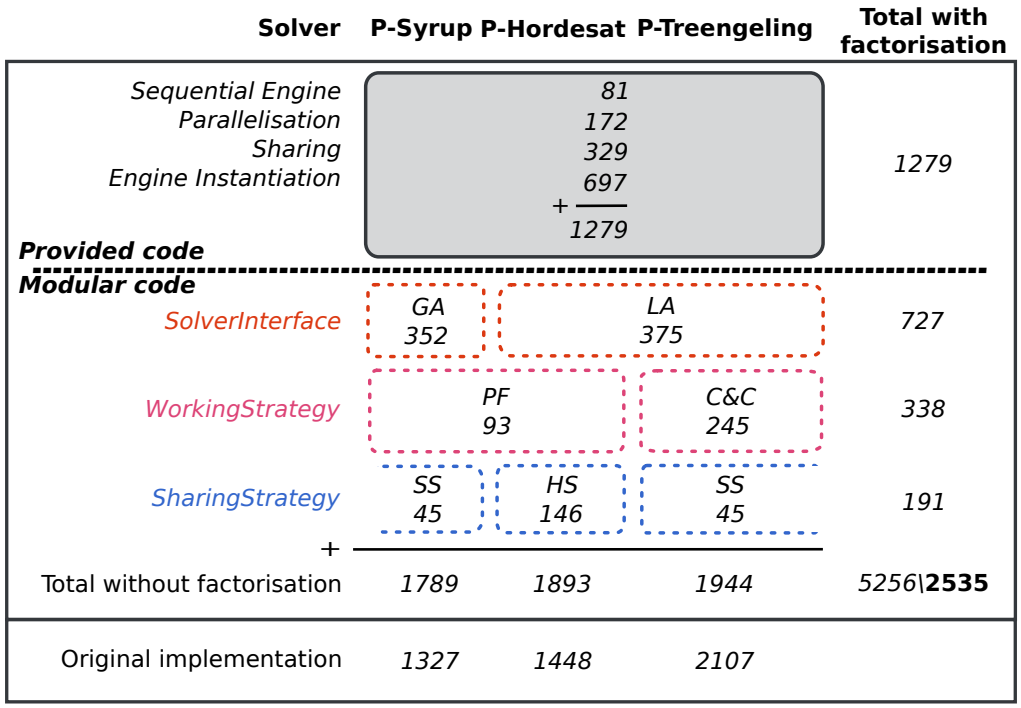


Figure 5.4: Analysis of the number of lines of codes required to implement the three mimicking solvers.

- The implementation of `P-HordeSat` needed 375 LoC for the `LingelingAdapter`, 93 LoC for the `Portfolio`, and 146 LoC for the `HordesatSharing` (HS). In total `P-HordeSat` required a total of 1893 LoC.

Even if the required number of LoC to implement the three solvers is 5256, thanks to the modularity of the framework we effectively needed only 2535 LoC. There are two ideas we want to highlight here: (1) the code provided by the framework allows users to focus on a particular component and clearly decreases the effort required to build new parallel SAT solvers; (2) when a component is implemented it can be reused for free, the more components are available more composition of solvers can be explored at a reduced cost.

Also as an indication the number of LoC of the original solvers (*i.e.,* `Syrup`, `Treengeling`, and `HordeSat`) is depicted in Figure 5.4. We can remark that the variation of LoC of the original solvers is varying in the same way as our implementations.

5.3.2 *Combining Existing Strategies*

We cannot say that something is reusable if it is not reused. Hence, while reusing the previously presented components, we quickly and easily built two completely new original solvers.

`P-Horgeling`. It is a `P-Treengeling`-based solver that shares clauses using the strategy of `Hordesat`. The implementation of this solver reuses the `LingelingAdapter`, `CubeAndConquer`, and `HordesatSharing` classes. To instantiate this solver we only needed a special parameterisation. Beside the modularity aspects, by this instantiation, we aimed to investigate the impact of a different sharing strategy on the overall performances of `P-Treengeling`.

`P-Syrupeling`. It is a portfolio solver that mixes cube-and-conquer of `Lingeling-Adapter`, and a portfolio of `GlucoseAdapter` solvers. Here, `Glucose` workers export unit and glue clauses [11] (*i.e.,* clauses with LBD equal to 2) to the other solvers. This last solver reuses the following components: `Lingeling`, `Glucose`, `Portfolio`, `Cube-AndConquer`, `SimpleSharing`. Only 15 LoC are required to build the parallelisation strategy tree. By the instantiation of this solver, we aimed to study the effect of mixing some parallelisation strategies.

5.4 Performance Evaluation

This section presents the results of experiments we realised using the solvers described in Section 5.2: `P-Syrup`, `P-Treengeling`, `P-HordeSat`, `P-Horgeling`, and `P-Syrupe-`

Solver	PAR-2	ALL (100)	UNSAT	SAT	CTI (#)
Syrup	346787.07	71	30	41	15h37 (69)
P-Syrup	**284486.06**	**78**	**32**	**46**	**13h18**
Treengeling	264235.02	82	32	50	20h55 (79)
P-Treengeling	**238850.07**	82	32	50	**14h12**
HordeSat	**304727.74**	**75**	31	**44**	15h05 (74)
P-HordeSat	311582.96	74	31	43	**14h19**

Table 5.5: Results of the different mimicking and mimicried solvers

ling. The goal here is to show that the introduction of genericness does not add an overhead *w.r.t.* the original solvers: Syrup[2], Treengeling[3], and HordeSat[4].

Experimental Setup. All the experiments have been executed on a parallel machine with 40 processors Intel Xeon CPU E7 - 2860 @ 2.27GHz, and 500 GB of memory. The used version of Linux and gcc are 4.1.15 and 4.9.2 respectively. We used the 100 instances of the parallel track of the SAT Race 2015[5]. All experiments have been conducted using the following parametrisations: each solver has been run once on each instance, with a timeout of 5000 seconds (as in the SAT competitions). We limited the number of involved CPUs to 36.

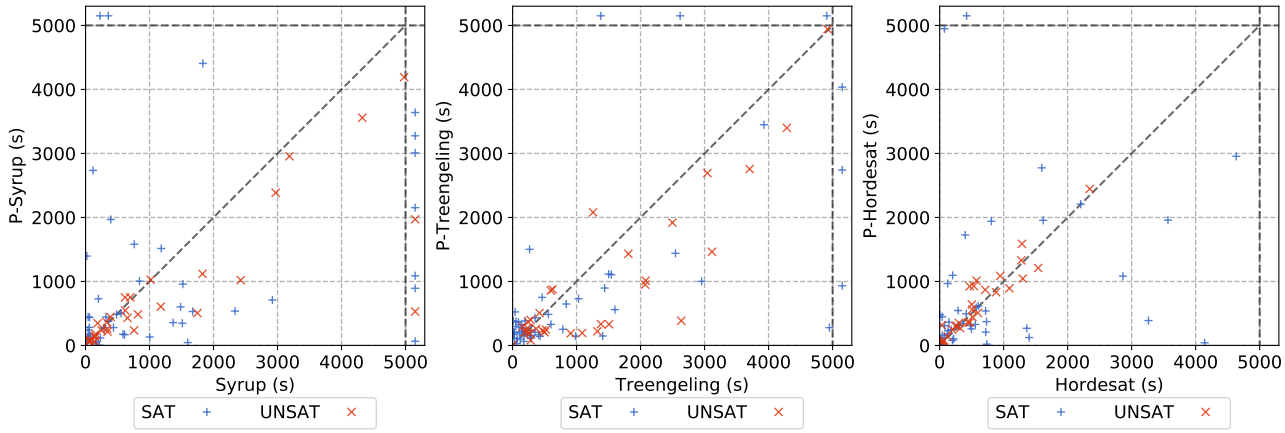
The results of these experiments are depicted in Table 5.5. The different columns represent respectively: the PAR-2 the measure used in the SAT competitions[6], number of total solved instances, UNSAT solved instances, SAT solved instances, and the cumulative execution time of the intersection (CTI) for instances solved by each pair of solvers (*i.e.,* the original and the Painless-based one) in hours.

Globally, these primary results show that our solvers compare well to the studied state-of-the-art solvers. We can deduce that the genericness offered by Painless does not impact the global performances. Moreover, on instances solved by both, the original solver and our implementation, the cumulative solving time is in our favour. A more detailed analysis is given for each solver hereafter.

5.4.1 *P-Syrup vs. Syrup*

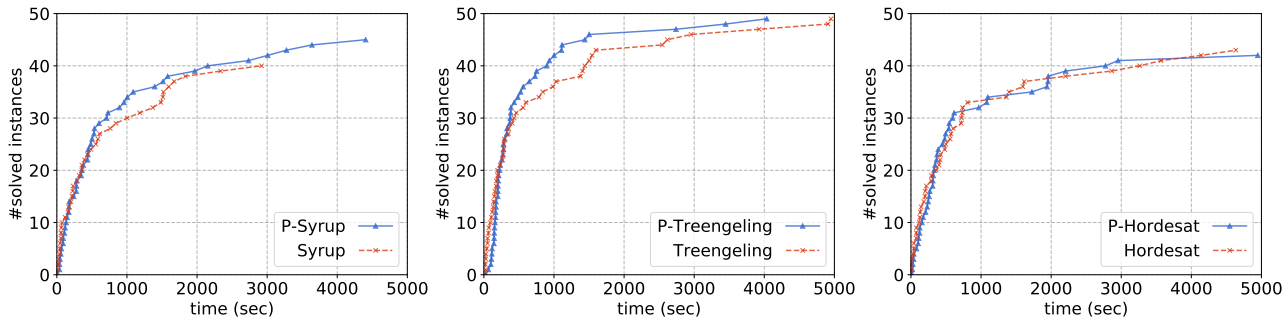
Our implementation of the Syrup's parallelisation strategy was able to solve more instances compared to Syrup (78 *vs.* 71), and is better in term of PAR-2 measure (346787.07 *vs.* 284486.06). This concerns both SAT and UNSAT instances as shown in the scatter plot of Fig-

[2] https://www.labri.fr/perso/lsimon/downloads/softwares/glucose-syrup.tgz
[3] http://www.fmv.jku.at/lingeling/lingeling-bbc-9230380-160707.tar.gz
[4] https://baldur.iti.kit.edu/hordesat/files/hordesat.zip
[5] https://baldur.iti.kit.edu/sat-race-2015/downloads/sr15bench-hard.zip
[6] PAR-k is the penalised average runtime, counting each timeout as k times the running time cutoff.



(a) `Syrup` **(b)** `Treengeling` **(c)** `HordeSat`

Figure 5.6: Scatter plots of `Painless`' solvers against state-of-the-art ones



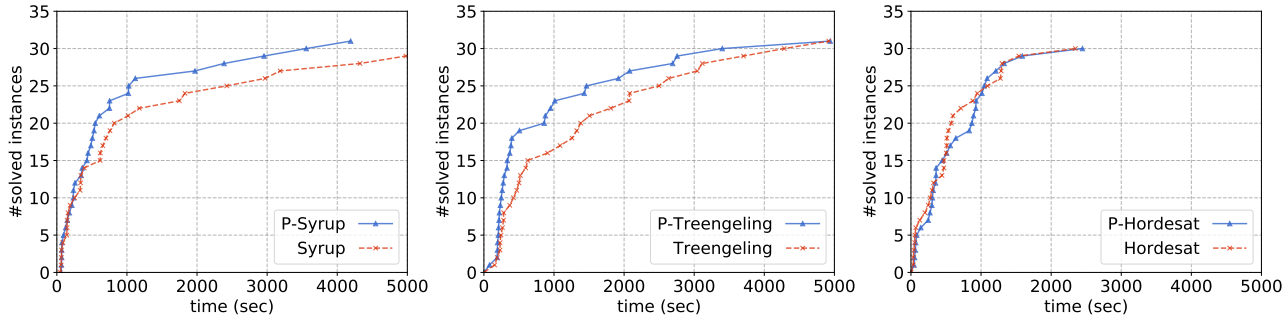
(a) `Syrup` **(b)** `Treengeling` **(c)** `HordeSat`

Figure 5.7: Mezcal plots of SAT instances of `Painless`' solvers against state-of-the-art ones

ure 5.6a and, in the mezcal plots[7] of Figure 5.7a and 5.8a. Even considering the 69 instances solved by both solvers, our implementation's CTI is 13h18 where `Syrup`'s is 15h37.

This gain is due to our careful design of the sharing mechanism that is decentralised and uses lock-free buffers. Indeed in `Syrup` a global buffer is used to exchange clauses, which requires import/export to use a unique lock, thus introducing a bottleneck. The absence of bottleneck in our implementation increases the parallel all over the execution, explaining our better performances.

[7]Classically, people in the community use cactus plots representing the time as a function of the number of solved instances. In this manuscript, we will present the plots upside down, *i.e.,* representing the number of instances as a function of time (*i.e.,* cumulative distribution function). Indeed, this representation seems more intuitive to most of the researchers (not familiar with the domain) to whom we have presented these plots. We call this representation mezcal plot because mezcal is a cactus alcohol which can make you see things upside down.



(a) `Syrup` **(b)** `Treengeling` **(c)** `HordeSat`

Figure 5.8: Mezcal plots of UNSAT instances of `Painless`'s solvers against state-of-the-art ones

5.4.2 *P-Treengeling vs. Treengeling*

Even, if `P-Treengeling` solves the same number of instances as `Treengeling` (82), our implementation is in average faster and has a better PAR-2 score (238850.07 *vs.* 264235.02). Moreover, the CTI calculated on the 79 instances solved by both solvers is 20h55 and 14h12, for `Treengeling` and `P-Treengeling` respectively. Our implementation was able to save more than 6h of computation, representing around 32% of the computational time of `Treengeling`. We can note that this speedup concerns both SAT and UNSAT instances as it is highlighted in Figure 5.7b and Figure 5.8b.

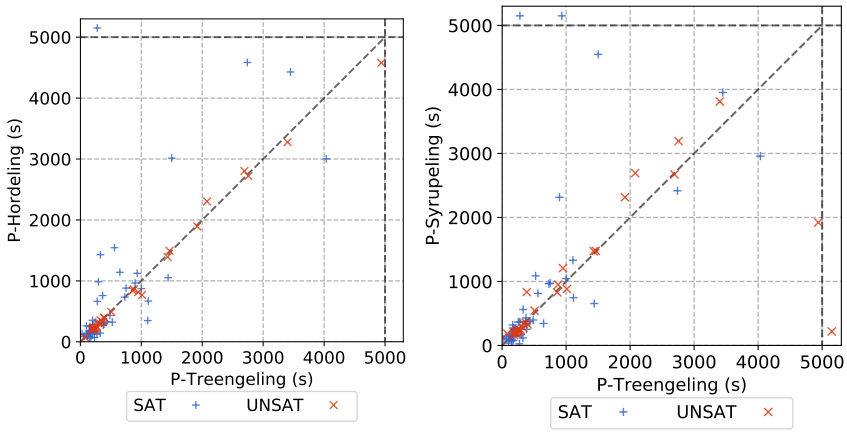
This speed up is due to our fine implementation of the cube-and-conquer strategy, thus increasing the real parallelism all over the execution and explaining our better performances.

5.4.3 *P-HordeSat vs. HordeSat*

Although `HordeSat` was able to solve 1 more instance than our tool (75 *vs.* 74), resulting in a better PAR-2 measure (304727.74 *vs.* 311582.96), we can say that both implementation are really close in terms of performance. Moreover scatter plot of Figure 5.6c, plus mezcal plots of Figure 5.8c and Figure 5.7c exhibit quit similar results for the two tools. For the 74 instances solved by both tools, our tool was a bit faster and used 46 minutes less as pointed out in Table 5.5.

These comparable results between `P-HordeSat` and `HordeSat` are normal since we just mimic the different strategies used in `HordeSat`. We can also conclude that `HordeSat` do not seem to suffer from concurrent programming.

Moreover, as the sharing strategy of `HordeSat` is mainly based on two parameters, namely the number of exchanged literals per round, and the sleeping time of the sharer by round, we think that a finer tuning of this couple of parameters for our implementation could improve the performances of our tool.



(a) P-Horgeling **(b)** P-Syrupeling

Figure 5.9: Scatter plots of the composed solvers against P-Treengeling

5.4.4 *Results of the Composed Solvers*

P-Horgeling solved 81 instances (49 SAT and 32 UNSAT), and P-Syrupeling solved 81 instances (48 SAT and 33 UNSAT). The scatter plots of the two strategies (Figure 5.9), show that these strategies are almost equivalent w.r.t. our best implementation namely P-Treengeling. These results allow us to conclude that the introduced strategies do not seem to add any additional value to the original one.

5.5 Assessment of Painless in the SAT Competition

Using the Painless framework, we have built different solvers to participate in different SAT competitions[8]. This section describes the different versions of our solver and presents results and analysis of the competition runs.

5.5.1 *SAT Competition 2017*

We now describe the overall behaviour of our competing instantiation [65] in the SAT competition 2017, namely P-MapleCOMSPS. Its architecture is highlighted in Figure 5.10.

P-MapleCOMSPS uses as sequential core engine MapleCOMSPS, the winner sequential solver of the main track of the SAT Competition 2016. It is based on MiniSat [38], and uses as decision the classical VSIDS [88] and the newly introduced LRB [70] heuristics. These heuristics are used in one-shot phases: first LRB, then VSIDS. We adapt this solver for the parallel context as follows: (1) we parametrised the solver to select either LRB or VSIDS for

[8] https://www.satcompetition.org/

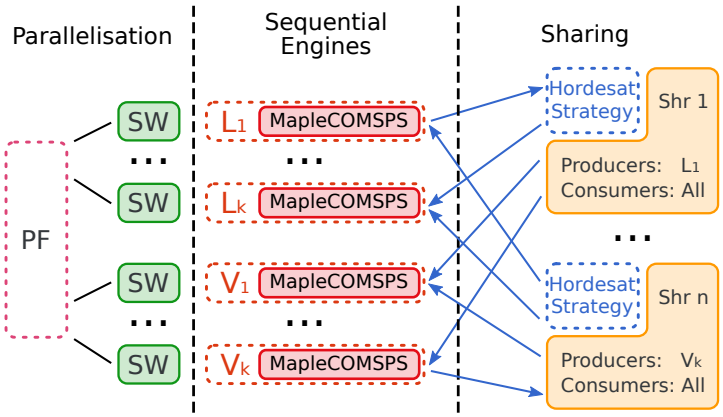


Figure 5.10: Architecture of `P-MapleCOMSPS`

all solving process (noted respectively, `L` and `V`); (2) we added callbacks to export and import clauses. The export is parametrised according to a LBD [11] threshold.

`P-MapleCOMSPS` reuses the `Portfolio` as working strategy, where the underlying core engines are either `L` or `V` instances. For each type of instances (*i.e.,* `L` and `V`), we apply a sparse random diversification similar to the one introduced in [14]. That is for each group of k solvers, the initial phase of a solver is randomly set according the following settings: every variable get a probability $1/2k$ to be set to false, $1/2k$ to true, and $1 - 1/k$ not to be set.

`P-MapleCOMSPS` can be parametrised to use n_cpus sequential solvers. Moreover, as some instances are huge and to avoid memory explosion, this limit can be decreased. First, one `MapleCOMSPS` solver is created, the memory print of this solver is raised (mem_solver). The portfolio is then launched using $used\_cpus = min(mem\_limit/mem\_solver, n\_cpus)$ sequential solvers.

In `P-MapleCOMSPS`, the sharing strategy used is `HordesatSharing`. We can note that `MapleCOMSPS` have 3 different databases to store learnt clauses as depicted in Figure 5.11. This clearly has a good impact on the performance of our `P-MapleCOMSPS` solver, and this allows us to exchange more clauses between the different workers as we know that they are carefully handled by the underlying sequential engine. Imported clauses are added to the database regarding the LBD they had when they were learnt.

The machine used to run the different solvers has 24 cores: 12 Dual Socket Xeon E5-2690 v3 (Haswell) @ 2.6GHz and 64 GB of RAM. Moreover, hyper-threading was activated porting the number of logical cores to 48. In the benchmark of the SAT Competition 2017 there is 350 instances both applicative and crafted. The used timeout in the SAT competitions is 5000 seconds.

The results of the podium solvers are depicted in Table 5.12. The columns represent: the PAR-2, the total solved instances, the number of UNSAT instances solved, the number of SAT instances solved, and the CTI. The measure used to rank the solvers in the SAT competition is PAR-2. This table has been produced using the gross results available on the competition's

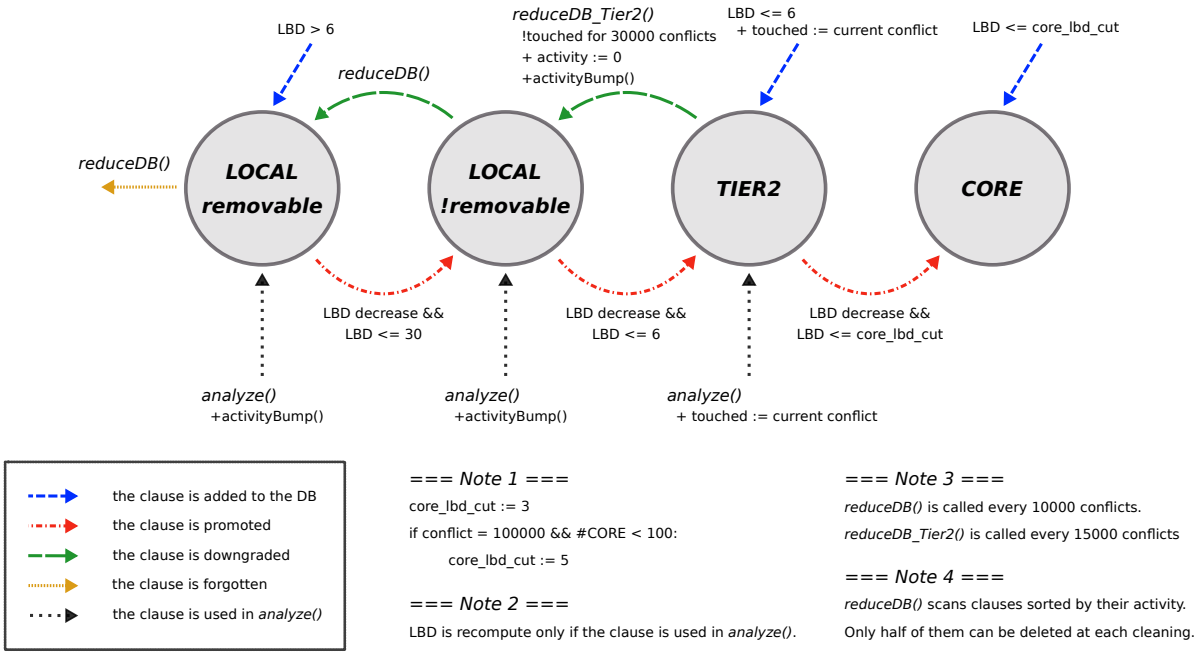


Figure 5.11: Life cycle of learnt clauses in the MapleCOMSPS solver

	Solver	PAR-2	ALL (350)	UNSAT	SAT	CTI (202)
1	Syrup	**1229297.26**	237	124	113	18h09
2	Plingeling	1266163.50	**239**	**129**	110	26h09
3	P-MapleCOMSPS	1368420.42	223	109	**114**	**17h58**

Table 5.12: Results of the podium solvers from the SAT Competition 2017

website[9].

An interesting point is that Plingeling solved more instances and UNSAT instances, P-MapleCOMSPS solved more SAT instances, but in average regarding PAR-2 Syrup is better. When considering the 202 instances solved by the three solvers, Syrup and P-MapleCOMSPS are clearly faster than Plingeling.

The first two Syrup and Plingeling were launched with 24 threads when P-MapleCOMSPS was launched with 48 threads. This has two bad impacts on our implementation. First, some instances are huge and many clauses are learnt during solving. We do not take this into account and this may conduct our implementation to reach memory limit on some instances. Moreover, it is not clear that hyper-threading would increase or decrease our performance due to cache cleaning and page defaults.

[9] https://baldur.iti.kit.edu/sat-competition-2017/results/parallel.csv

	Solver	PAR-2	ALL (400)	UNSAT	SAT	CTI (197)
1	P-MapleCOMSPS	**1397524.67**	**275**	**122**	153	**11h47**
2	Plingeling	1410158.33	**275**	119	**156**	18h28
3	CryptoMiniSat	1755049.35	241	88	153	36h59

Table 5.13: Results of the podium solvers from the SAT Competition 2018

5.5.2 *SAT Competition 2018*

We now describe the overall behaviour of our competing instantiation [68] in the SAT Competition 2018, namely P-MapleCOMSPS. Its architecture is the same as the one that participated in the SAT competition in 2017 (see Figure 5.10). The only difference is the version of the underlying MapleCOMSPS.

Mainly there were two changes compared to the version we used in 2017, both concerning the core engine MapleCOMSPS:

- the version of MapleCOMSPS we used in 2018 embedded a Gaussian elimination preprocessor (see Section 2.2.3). Only one of the workers of our portfolio does this preprocessing.
- In MapleCOMSPS learnt clauses with falsified literal at level 0 are shrinked. This allows the solver to see clauses real size, and prevents useless check. Moreover, this shrink is also done on imported clauses. Their LBD is then updated to the minimum between the original LBD and the size of the clause after shrinking. This allows us to better introduce clauses in the life cycle of learnt clauses depicted in Figure 5.11.

The machine used to run the different solvers has 24 cores: 12 Dual Socket Xeon E5-2690 v3 (Haswell) @ 2.6GHz and 64 GB of RAM. Moreover, hyper-threading was activated porting the number of logical cores to 48. In the benchmark of the SAT Competition 2018 there are 400 instances both applicative and crafted. The used timeout in the SAT competitions is 5000 seconds.

The results of the podium solvers are depicted in Table 5.13. The columns represent: the PAR-2, the total solved instances, the number of UNSAT instances solved, the number of SAT instances solved, and the CTI. The measure used to rank the solvers in the SAT competition is PAR-2. This table has been produced using the gross results available on the competition's website[10].

First we have to say that on the official results AbcdSAT is third. In fact, when analysing the results one can observe that AbcdSAT produced wrong results. Indeed, it answered UNSAT for one instance (namely Cake_8_16.cnf) that was proven SAT by Plingeling.

[10] https://sat2018.forsyte.tuwien.ac.at/results/parallel18.csv

P-MapleCOMSPS and Plingeling solved the same number of instances, but P-MapleCOMSPS had a better PAR-2 measure. Both solvers were really close in terms of performance. One interesting point is when considering the 197 instances solved by both solvers, P-MapleCOMSPS is clearly faster and has a CTI of 11h47 while the one of Plingeling is 18h28. Around 36% of time is saved by P-MapleCOMSPS on these instances.

We think that the difference between the performance of two versions (of 2017 and 2018) are due to: first, instances used in the 2017 and 2018 are not the same and this can have a real impact on solvers efficiency. Secondly, this year we used only 24 threads, and garbage collection of MapleCOMSPS was better, so we think that this allows us to share fewer clauses and manage them a better way.

5.6 Conclusion

Implementing and testing new strategies for parallel SAT solving has become a challenging issue. Any new contribution to the domain faces the following problems: (1) concurrent programming requires specific skills, (2) testing new strategies required a prohibitive development of a complete solver (either built from scratch or an enhancement of an existing one), (3) an implementation often allows to test only a single composition policy and avoids the evaluation of a new heuristic with different versions of the other mechanisms.

To tackle these problems we proposed Painless, a modular, generic and efficient framework for parallel SAT solving. We claimed that its modularity and genericness allow the implementation of basic strategies, as well as new ones and their combination with minimal effort and concurrent programming skills.

We have shown the genericness of Painless by the implementation of strategies present in state-of-the-art solvers: Syrup, Treengeling, and HordeSat. Second, we highlighted its modularity by reusing developed complements to derive, easily, new solvers that mix strategies.

We have also shown that the instantiated solvers are as efficient as the original ones (and even better), by conducting a set experiments using benchmarks of the SAT Race 2015. Moreover, using Painless we were able to build two versions of the P-MapleCOMSPS solver, that have been ranked 3rd and 1st in the parallel track of the SAT Competition respectively in 2017 and 2018.

Chapter

6

Implementing Efficient Divide-and-Conquer

Contents

Mainly, two classes of parallelisation techniques have been studied: divide-and-conquer (D&C) and portfolio (see Chapter 4). Although divide-and-conquer approaches seem to be the natural way to parallelise SAT solving, the outcomes of the parallel track in the annual SAT competitions[1] show that the most efficient state-of-the-art parallel SAT solvers are portfolios.

The main problem of divide-and-conquer based approaches is the search space division so that load is balanced over solvers, which is a hard theoretical problem. Hence heuristics are used, and solvers compensate non-optimality by enabling *dynamic load balancing*. However, state-of-the-art SAT solvers appropriately dealing with these aspects are hardly adaptable to various strategies than the ones they have been designed for [7, 18, 9]. Hence, it turns out to be very difficult to make fair comparisons between techniques (*i.e.,* using the same basic implementation). Thus, we believe it is difficult to conclude on the (non-) effectiveness of a technique with respect to another one and this may lead to premature abortion of potential good ideas.

[1] https://www.satcompetition.org/

This chapter tries to solve these problems by proposing a simple, generic, and efficient divide-and-conquer component [67] build on top of the `Painless` framework (see Chapter 5). This component eases the implementation and evaluation of various strategies, without any compromise on efficiency.

This chapter is organised as follows: Section 6.1 explains the mechanism of divide-and-conquer we have implemented in `Painless`. Section 6.2 explains the different heuristics we have implemented. Section 6.3 analyses the results of our experiments. Section 6.4 concludes this chapter and discusses future works.

6.1 IMPLEMENTATION OF A DIVIDE-AND-CONQUER

This section presents the divide-and-conquer component we have built on top of `Painless`. We describe the generic divide-and-conquer component mechanisms. Finally, we detail the different heuristics we have implemented using this component.

To implement divide-and-conquer solvers with `Painless`, we define a new component based on a master/slave architecture.

Figure 6.1 gives an overview of our architecture. It shows several entities:

- the *master* is a component responsible for the management of the `D&C` which is an instance of the `WorkingStrategy` class. It is also responsible of the management of a work queue (noted `wq` in Figure 6.1).
- The *sequential workers* (*i.e.,* slaves) are components responsible for solving the subspaces and are instances the `SequentialWorker` class.
- Each worker operates a sequential solver. The panel of solvers form the *sequential engines* concept component. Each solver is an instance of the `SolverInterface` class.
- An instance of the *sharing* concept component allows workers to share clauses. Sharing is based on the classes `SharingStrategy` and `Sharer` as described in Section 5.1.

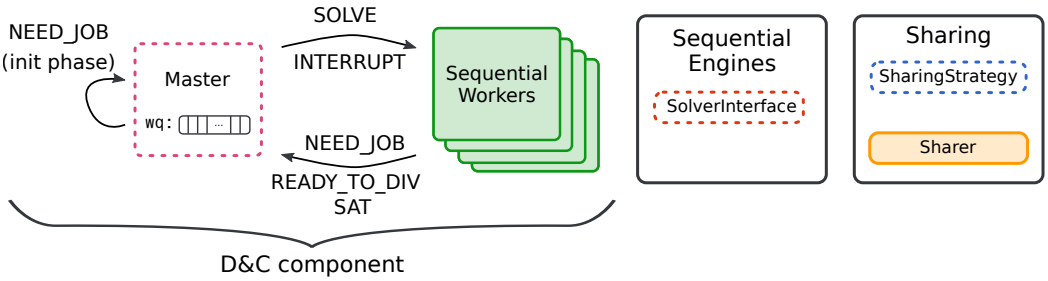


Figure 6.1: Architecture of the `Painless`'s divide-and-conquer component

The master and the workers interact asynchronously by means of events. In the initialisation phase, the master may send asynchronous events to himself too.

6.1.1 *Slaves Life Cycle*

A slave may be in three different states: *idle*, *work*, and *work_interrupt_requested*. Initially, it is *idle* until it receives a `SOLVE` event from the master. Then, it moves to the *work* state and starts to process its assigned subspace. It may while working:

- find a solution, then it emits a `SAT` event to the master and moves back to *idle* state;
- end processing the subspace, with an UNSAT result, then it emits a `NEED_JOB` event to the master, and moves back to *idle* state;
- receive an `INTERRUPT` event from the master, then, it moves to the *work_interrupt_requested* state and continues its processing until it reaches a stable state[2] according to the underlying sequential engine implementation. Then, it sends a `READY_TO_DIV` event to the master prior to move back to *idle* state.

6.1.2 *Master Life Cycle*

The master (1) initialises the `D&C` component; (2) selects targets to divide their search spaces; and (3) operates the division along with the relaunch of the associated solvers. These actions are triggered by the events `INIT`, `NEED_JOB` and `READY_TO_DIV`, respectively. For the remainder of this section we consider a configuration with N workers.

The master can be in two states: either it is *sleeping*, or it is currently *processing* an incoming event. Initially the master starts a first solver on the whole formula by sending it a `SOLVE` event. It then generates $N - 1$ `NEED_JOB` events to himself. This will provoke the division of the search space in N subspaces according to the implemented policy. At the end of this initialisation phase, it returns to its sleeping state. At this point, all workers are solving their subspaces.

Each time a worker needs a job, when it has encountered an UNSAT situation, it notifies the master with a `NEED_JOB` event.

All over its execution, the master reacts to the `NEED_JOB` events as follows:

(1) if the work queue contains jobs, a job is popped in the queue. The master sends a `SOLVE` event to the idle worker and notify it to treat the job that has been popped from the queue.

[2]For instance, in `MiniSat`-based solvers, a stable state could correspond to the configuration of the solver after a restart.

(2) If the work queue is empty, it selects a target using the current policy[3], and requests this target to interrupt by sending an `INTERRUPT` event. Since this is an asynchronous communication, the master may process other events until it receives a `READY_TO_DIV` event from the target.

(3) Once it receives a `READY_TO_DIV` event, the master proceeds to the effective division of the subspace of the worker which emitted the event. The worker that emitted the event and the one that requested a job are then invited to solve their new subspaces through the send of a `SOLVE` event.

The master may receive a `SAT` event from its workers. It means that a solution has been found and the whole execution must end. When a worker ends in an UNSAT situation, it makes a request for a new job (`NEED_JOB` event). When the master has no more division of the search space to perform, it states the formula is UNSAT.

We can note that it is relevant to have a queue with a max size forced to 0, meaning the work queue is never used. In the case where the used division heuristic is look back (*i.e.,* based on gathered statistics) waiting until the last moment can create more suitable divisions. Indeed, the statistics used to divide are more recent and updated.

6.2 IMPLEMENTED HEURISTICS

The divide-and-conquer component presented in the previous section is generic enough to allow the implementation of any of the state-of-the-art strategies presented in Section 4.2. We selected some strategies to be implemented, keeping in mind that at least one of each family of mechanisms should be retained:

(M1) Techniques to divide the search space (Section 4.2.1): we have implemented the guiding path method which is the most used in the literature. Technically it is implemented by the use of assumptions. Since we want to be as generic as possible, we have not considered techniques adding constraints to the formula (because they require complex to implement tagging mechanisms to enable clause sharing).

(M2) Choosing division variables (Section 4.2.2): the different division heuristics we have implemented in the `MapleCOMSPS` solver [71] and in the `Glucose` [11] solver are: VSIDS, number of flips, and propagation rate.

(M3) Dynamic load balancing (Section 4.2.3): a work queue and work-stealing mechanisms were implemented to operate dynamic load balancing. The master looks in the work queue and if it is empty it selects targets using a FIFO policy (as in `Dolius` [7]) moderated by a minimum computation time (2 seconds) for the workers in order to let these acquire a sufficient knowledge of the subspace.

[3]This policy may change dynamically over the execution of the solver.

(M4) Exchange of learnt clauses (Section 4.2.4): on any of the strategies we implemented it is not restricted. This allows to reuse any of the already off-the-shelf strategies provided by `Painless`.

(M5) Workers' knowledge management: another important issue deals with the way new subspaces are allocated to workers. There are three strategies:

- *Reuse:* the worker reuses the same object solver all over its execution and the master feeds it with guiding paths. The worker gathered global experience and keep it all over its life.
- *Clone:* each time a new subspace is assigned to a worker, the master clones the object solver from the target and provides the copy to the idle worker. Thus, the idle worker will implicitly benefit from the local knowledge (VSIDS, locally learnt clauses, etc.) of the target worker.

(M6) Underlying sequential solvers: any of sequential solvers can be used, so we can reuse off-the-shelf adapters provided by `Painless`. At least for look back heuristics probes should be added to the solver in order to get statistics. This has been done for `MapleCOMSPS` and `Glucose`.

Hence, our `Painless`-based `D&C` component can thus be instantiated to produce solvers over six orthogonal axes: (M1) technique to divide the search space; (M2) technique to choose the division variables; (M3) dynamic load balancing strategy; (M4) the sharing strategy; (M5) the strategy to manage workers' sequential knowledge; and (M6) the used underlying sequential solver.

There exists at least 48 possible combinations of implemented heuristics. Hence, we have to choose some. Table 6.2 summarises the implemented `D&C` solvers we have used for our experiments in the next section. They are varying over two axes: (M2) and (M5)[4]. Three of these solvers are original implementations that have never been used, to our knowledge, in the literature. These solvers are highlighted in bold in Table 6.2.

All these solvers are based on `MapleCOMSPS`, and sharing all learnt clauses which LBD ≤ 4 (this value has been experimentally deduced). Sharing is done using the `SimpleSharing` strategy defined in Chapter 5. Moreover, as the division heuristics used are all look back we choose to set the limit max size of the work queue to 0 (*i.e.,* not using it).

6.3 Evaluation of the Divide-and-Conquer Component

This section presents the results of experiments done with the six `D&C` solvers we presented in Section 6.2. We also did comparative experiments with state-of-the-art `D&C` solvers (`Treengeling` [18] and `MapleAmpharos` [90]).

[4]Indeed, these two axes seem to be the most relevant ones.

(M5) \ (M2)	VSIDS	Number of flips	Propagation Rate
Reuse	`P-Reuse-VSIDS`	`P-Reuse-Flips`	`P-Reuse-PR`
Clone	**`P-Clone-VSIDS`**	**`P-Clone-Flips`**	**`P-Clone-PR`**

Table 6.2: The `D&C` solvers we use for experiments in this chapter

`Treengeling` is a cube-and-conquer solver based on the `Lingeling` sequential solver. `MapleAmpharos` is an adaptive divide-and-conquer based on the solver `AmPharoS` [9], and uses `MapleCOMSPS` as sequential solver. Comparing our new solvers with state-of-the-art ones (*e.g.,* not implemented on `Painless`) is a way to assess if our solutions is competitive despite the genericity introduced by `Painless` and ad-hoc optimisations implemented in other solvers.

All experiments were executed on a multi-core machine with 12 physical cores (2 x Intel Xeon E5645 @ 2.40 GHz), and 64 GB of memory. Hyper-threading has been activated porting the number of logical cores to 24. The used version of Linux and gcc is 4.9.0 and 8.2.0 respectively. We used the 400 instances of the parallel track of the SAT Competition 2018[5]. All experiments have been conducted using the following settings:

- each solver has been run once on each instance with a time-out of 5000 seconds (as in the SAT Competition);
- the number of used cores is limited to 23 (the remaining core is booked to the operating system);
- 40 instances that were trivially solved by a solver (at the preprocessing phase) were removed. Indeed in this case the `D&C` component of solvers is not enabled, and these instances are then irrelevant for our case study.

Results of these experiences are summarised in Table 6.3. The different columns represent respectively: the total number of solved instances; the number of UNSAT solved instances, the number of SAT solved instances, and the PAR-2 score.

6.3.1 *Comparing the Implemented Divide-and-Conquer*

Figure 6.4 presents the mezcal plot of the performances of the different `D&C` solvers. These differ in two orthogonal axes: the used division heuristic, and the used subspace allocation technique. We analyse here each axe separately.

When considering the allocation technique (clone *vs.* reuse), we can observe that the cloning based strategy is globally more efficient, even if it has a supplementary cost (due to the cloning

[5] http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip

Solver	PAR-2	ALL (360)	UNSAT	SAT
P-Clone-Flips	**1732696.65**	198	**87**	111
P-Clone-PR	1871614.48	183	73	110
P-Clone-VSIDS	1880281.54	183	77	106
P-Reuse-Flips	1796426.72	190	83	107
P-Reuse-PR	1938621.48	180	72	108
P-Reuse-VSIDS	1868619.43	184	75	109
MapleAmpharos	2190680.55	153	29	**124**
Treengeling	1810471.56	**200**	84	116

Table 6.3: Results of the different D&C solvers

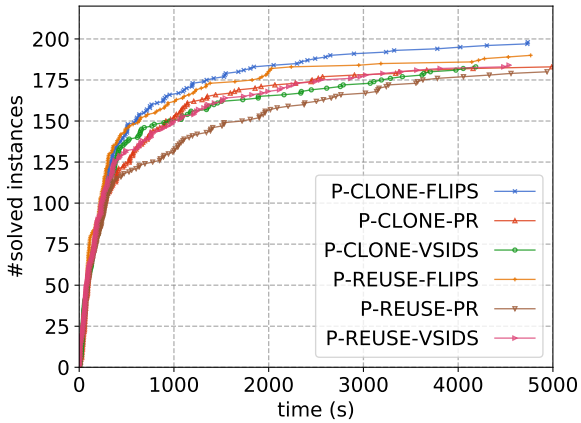


Figure 6.4: Mezcal plot of our different D&C based solvers

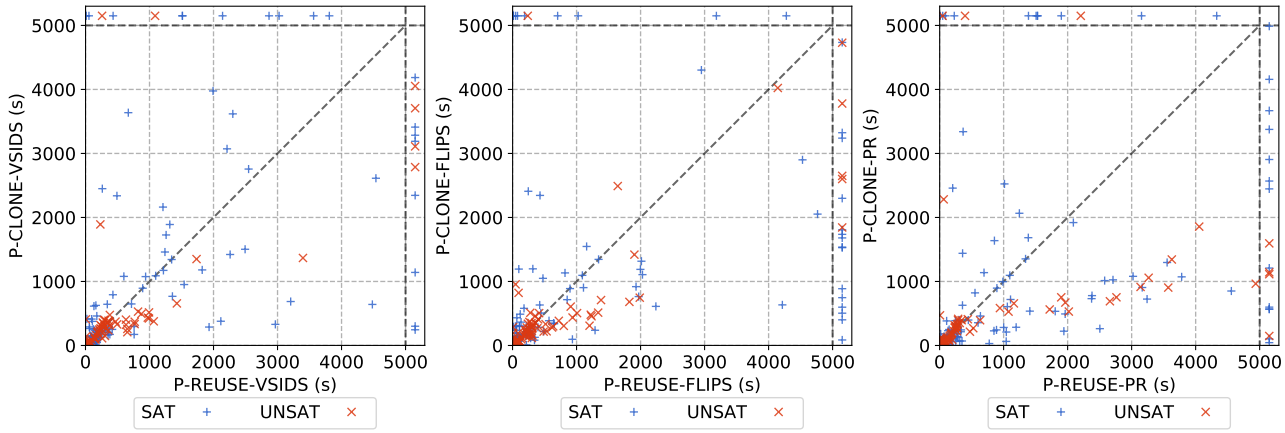
phase). The scatter plots of Figure 6.5 confirm this observation: most plots are below the diagonal, showing evidence of a better average performance of clone *vs.* reuse. We believe this is due to the local knowledge (*e.g.,* learnt clauses, VSIDS, phase saving) that is implicitly shared between the (cloned) workers.

When considering the division heuristics (VSIDS, number of flips, and propagation rate), we observe that the number of flips based approach is better than the two others. Both, by the number of solved instances and the PAR-2 measure. This is particularly true when considering the cloning based strategy. VSIDS and propagation rate based solvers are almost identical.

6.3.2 *Comparison with State-of-the-Art Divide-and-Conquer*

Figure 6.6 shows a mezcal plot comparing our best implemented divide-and-conquer solver (*i.e.,* P-CLONE-FLIPS) against Treengeling and MapleAmpharos.

The MapleAmpharos solver seems to be less efficient than our tool, and solves fewer instances. When considering only 123 instances that both solvers were able to solve, we can



(a) VSIDS **(b)** Number of flips **(c)** Propagation rate

Figure 6.5: Scatter plots of divide-and-conquer reusing *vs.* cloning solvers

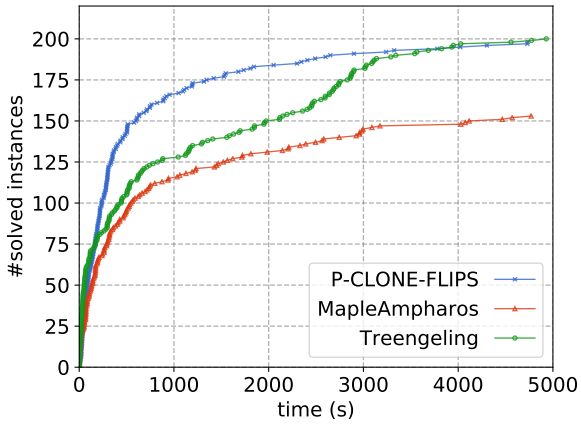


Figure 6.6: Mezcal plot of our best instantiated `D&C` and state-of-the-art solvers

calculate the cumulative execution time of the intersection (CTI) for `MapleAmpharos` and `P-Clone-Flips`: it is, respectively, 24h33min and 14h34min.

Although our tool solves 2 fewer instances than `Treengeling`, it has better PAR-2 measure. The CTI calculated on the 169 instances solved by both solvers is 49h14min and 22h23min, respectively for `Treengeling` and `P-Clone-Flips`. We can say that even if both solve almost the same number of instances, our `D&C` solver is faster. We clearly observe this phenomenon in Figure 6.6.

Thus, in addition to highlight the performance of our instantiation, this shows the effectiveness of the flip-based approach with respect to the well-proven cube-and-conquer strategies.

6.4 Conclusion and Discussion

This chapter proposed an efficient implementation of several parallel SAT solvers using the divide-and-conquer (D&C) strategy that handle parallelism by performing successive divisions of the search space.

Such an implementation was performed on top of the Painless framework allowing to easily deal with variants of strategies. Our Painless-based implementation can be customised and adapted over six orthogonal axes:

- (M1) technique to divide the search space;
- (M2) technique to choose the division variables;
- (M3) dynamic load balancing strategy;
- (M4) the sharing strategy;
- (M5) the workers' knowledge management;
- and (M6) the used underlying sequential solver.

This work shows that we now have a modular and efficient framework to explore new D&C strategies along these six axes. We are then able to make a fair comparison between numerous strategies.

Among the numerous solvers we have available, we selected six of them for performance evaluation. Charts are provided to show how they competed, but also how they cope face to D&C state-of-the-art solvers.

This study shows that the flip-based approach in association with the clone policy outperforms other strategies whatever the used standard metric is. Moreover, when compared with the state-of-the-art D&C-based solvers, our two best solvers show to be very efficient and allow us to conclude the effectiveness of our modular platform based approach with respect to the well-competitive D&C solvers.

It would be interesting to explore more implementations of heuristics and strategies such as: look ahead division heuristics coupled with the use of the work queue, rich/poor paradigm for dynamic load balancing, and impact of sharing in the context of divide-and-conquer.

There are still many works to do to improve D&C paradigm. First, as divisions are done using heuristics, designing a generally more efficient one seems to be a hard task. A possibility to select the best heuristics for the current formula is the use of machine learning process to rank variables. This is something we are currently testing in collaboration with researchers from Waterloo University in Canada. A second possibility is the exploration of global restarts, as in sequential, sometimes the overall splitting tree could be rebuilt using knowledge gathered during the search.

Chapter

7

Exploiting the Modularity of SAT Instances for Information Sharing

Contents

The purpose of the work presented in this chapter is to better understand the impact of learnt clauses on solvers' efficiency. A better quantification of the relevance of learnt clauses is important particularly in the parallel context to select exchanged clauses such contributing to solve challenge C2.

More precisely the goal is to produce relevant learnt clauses by exploiting the *modularity* of SAT instances. Indeed, in this thesis we have focused on complete solving, which is more efficient on instances encoding problems in real life. It is well admitted that this type of instances have a particular notable structure, explaining why some heuristics such as VSIDS [88] or phase saving [95] work well. One way to highlight this modularity is to represent the formula as a graph and to run a community detection algorithm on the generated graph [3].

Based on community detection, authors of [4] have proposed to preprocess the formula to produce "good" clauses. These clauses are then added to the original formula and allow a speed up when solving the enhanced instance *w.r.t.* the time required to solve the original formula. This chapter analysis the behaviour of this mechanism to go further and produce more of these "good" clauses.

First, Section 7.1 introduces ways to represent SAT instances as graphs. Section 7.2 presents the method used for community detection. Section 7.3 presents related works based on the graph structure of SAT instances. Section 7.4 proposes a novel algorithm to generate more "good" clauses. The impact of these "good" clauses is shown in the sequential and parallel contexts in Section 7.5 and Section 7.6 respectively. Finally, Section 7.7 concludes this work.

7.1 FROM SAT INSTANCES TO GRAPHS

This section presents different possible graph representations of SAT instances. All the representations presented in the section are depicted in Figure 7.1. These representations are then discussed.

7.1.1 *Hypergraph*

The natural way to represent the structure of a CNF formula is the construction of an *hypergraph*. An hypergraph is a graph where edges (called *hyperedges*) can link more than two nodes. Its building for a CNF formula φ gives the hypergraph whose nodes represent the variables of φ and hyperedges are clauses of φ. Each clause of size k generate an hyperedge having k endpoints being the variables of this clause.

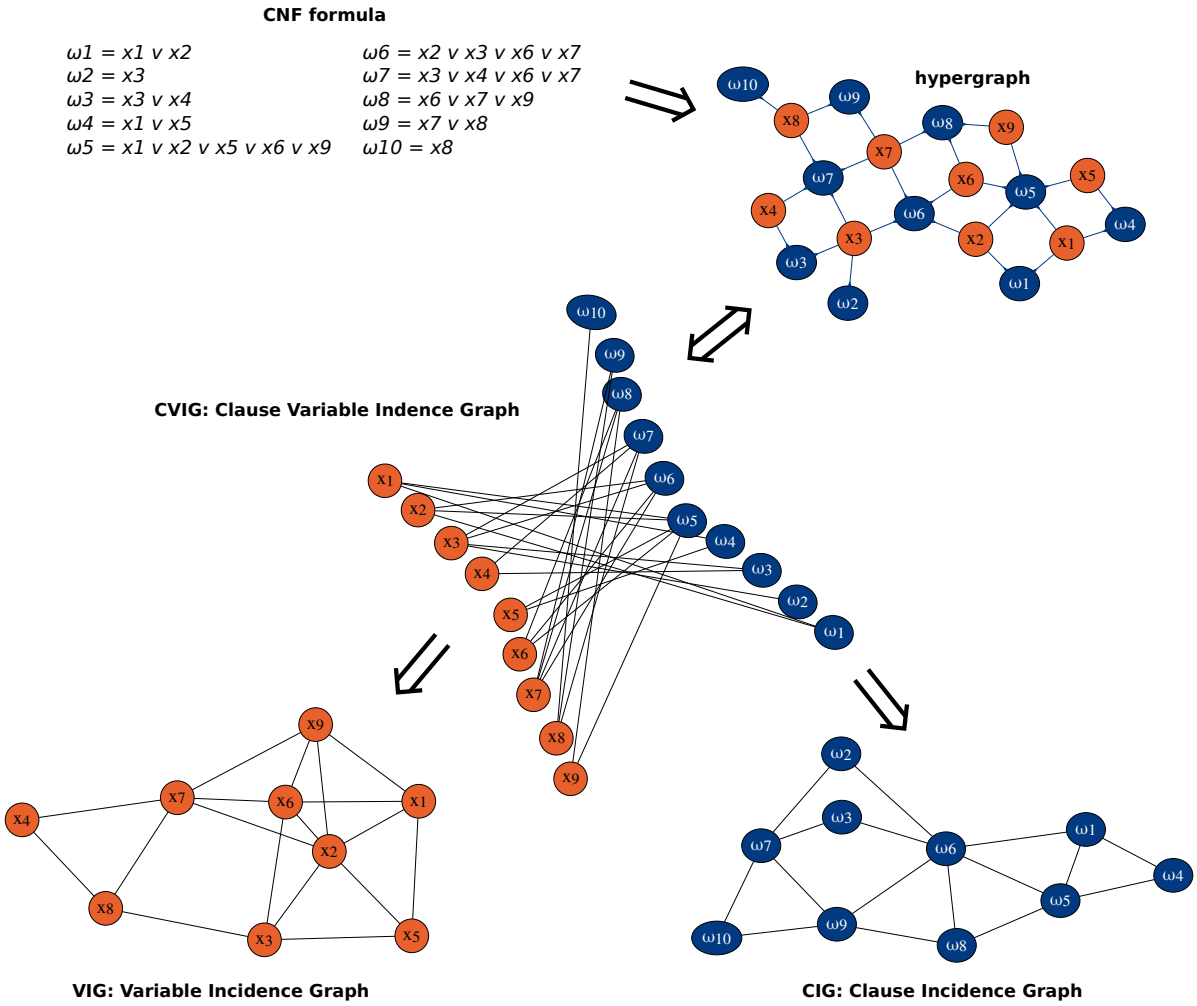


Figure 7.1: From CNF formula to different graph representations

7.1.2 *Clause Variable Incidence Graph*

An hypergraph can be represented as a *bigraph*. A bigraph is a graph for which set of nodes can be decomposed into two disjoint parts such that there exist no edges linking two nodes belonging to the same part. When transforming the hypergraph of a CNF formula into the associated bigraph, one creates the *clause variable incidence graph (CVIG)* [3] of this instance. Let φ be a formula, the CVIG is the bigraph with nodes representing the set of its variables of φ and the set of clauses of φ. There exists a link between a clause and a variable if the variable is contained in the clause.

7.1.3 *Variable Incidence Graph*

In order to get back to a simple graph, a bigraph can be projected over one of its two parts of nodes. In our case we can project this graph on the nodes representing variables or clauses.

When projecting on variables, we generate the so-called *variable incidence graph (VIG)* [3]. Let φ be a formula, the VIG of φ is the graph whose nodes represent the variables of φ, and there exists an edge between two variables if they both appear in a clause $\omega \in \varphi$.

With this construction the information of the size of clauses, and the number of time a pair of variables is part of the same clause are lost. A possible solution to keep this information is to distribute for each clause a weight of 1 over each of its variables. Such a weight function is defined as follows:

$$w(x_1, x_2) = \sum_{\substack{\omega \in \varphi \\ x_1, x_2 \in \omega}} \frac{1}{\binom{|\omega|}{2}} \tag{7.1}$$

7.1.4 *Clause Incidence Graph*

In the same way, when projecting on clauses, we generate the so-called *clause incidence graph (CIG)*. Let φ be a formula, its CIG is the graph with nodes representing clauses of φ and there exists an edge between two clauses if they share a variable x of φ. Here the lost information is the number of time pairs of variables appears in the same clauses. We can keep track of this information by defining, for instance, the following weight function, where $\mathtt{cls}(x)$ returns the set of clauses in which variable x appears:

$$w(\omega_1, \omega_2) = \sum_{\substack{x \in \omega_1 \cap \omega_2 \\ \omega_1, \omega_2 \in \varphi}} \frac{1}{\binom{|\mathtt{cls}(x)|}{2}} \tag{7.2}$$

7.1.5 *Discussing the Different Representations*

Since instances treated by SAT solvers can have up to millions of variables and tens millions of clauses, the selected graph representation can be very huge and should be carefully chosen. Considering the number of nodes, the following graphs are presented in increasing order of their likeliness to have more nodes: VIG, CIG, CVIG.

Also smaller graphs (*i.e.,* VIG and CIG) have less information due to the projection. The targeted graph should be considered depending on the desired level of precision. Since, in the case of SAT solving this information is often used heuristically it is not important to derive less precise information. For the remaining of the chapter we will focus on the VIG which is the smallest in terms of the number of nodes.

7.1.6 *Literal-based Representation*

It is also possible to represent a formula φ using its literals instead of variables. As for a variable x, the positive and the negative literals are linked one can add an edge between them.

This edge can be seen as the clause $(x \vee \neg x)$. We can note that adding this clause to the original formula does not change its satisfiability: $\varphi \equiv \varphi \wedge (x \vee \neg x)$, it only insures Boolean consistency. Using the same approach, as the one described above, we can define similar graphs based on literals.

7.2 Community-based Approach

Our goal is to characterise the quality of learnt clauses using a static measure. In SAT instances, some variables are more constraints together (linked by the clauses). A manner to characterise this is the *modularity* which is introduced in the first part of this section. Afterwards, we describe the Louvain method used to perform community detection. To get more complete overview of community structure detection in graphs the reader can refer to [41].

7.2.1 *Modularity*

The *modularity* is a measure that has been introduced by [91]. For a given graph G, and a partition $P = \{P_1, \ldots, P_n\}$ of nodes of G, the modularity is defined as follows:

$$\mathcal{Q}(G, P) = \sum_{P_i \in P} \frac{\sum_{x,y \in P_i} w(x,y)}{\sum_{x,y \in V} w(x,y)} - \left(\frac{\sum_{x \in P_i} \deg(x)}{\sum_{x \in V} \deg(x)} \right)^2 \tag{7.3}$$

Our goal is to find a partition of the nodes of G maximising the modularity. The optimal modularity of the graph G is the maximal modularity, for any possible partition P of its nodes vertices: $Q(G) = \max\{Q(G,P)|P\}$. The optimal modularity of a graph is in the range $[0,1]$. Also finding this best partition is an NP-complete problem.

The general idea of the modularity described in this section is that nodes of the same part (community) have more probability of being linked, compared to the probability of being linked with outer nodes.

7.2.2 *Louvain Method*

A greedy and efficient algorithm, returning an approximated lower bound for the modularity, is the Louvain method [23], it is described in Algorithm 7. This algorithm has a time complexity of $O(n \log n)$, where n is the number of nodes of the graph.

It is a bottom-up algorithm. This means that at the beginning, each node forms a community (Line 3) and over time these communities can be merged to maximise the modularity. For each node we look for the greediest move to improve the measure (Line 6). According to this move, the node is changed or not of community (Line 7) change according to this choice. The process continues while the improvement of the metric is under a certain threshold ϵ. When

it is not the case any more there is a change of granularity used (Line 8). A new graph is considered where nodes are the current community. The idea here is that nodes will not be moving one by one but by packages. The procedure stop if even with a change of granularity the measure does not continue to increase sufficiently (Line 2).

```
1 function Louvain(G: a graph)
    /* returns P a partition of nodes of G                    */
2   while improvement > ϵ do
3     P ← G.nodes
4     while improvement > ϵ do
5       foreach n in G.nodes do
6         k ← find best move for n
7         move n to P_k
8     G ← changeGranularity(G, P)
9   return P
```

Algorithm 7: Louvain method

7.3 RELATED WORKS

This section presents related works exploiting the modularity of SAT instances based on a graph representation.

7.3.1 *Highlighting*

The modularity is a way to highlight the structure of CNF formulas. This modularity is computed by transforming SAT instances into graphs. Authors of [3] show that VIG of instances encoding real world problems exhibit a particularly high modularity (around 0.8), contrariwise random instances have a very small one (less than 0.3).

Since the Louvain method complexity depends on the number of nodes of the graph, it is preferred to apply community detection on the VIG. Authors of [4] have proposed a technique to compute the communities of clauses from the VIG. They first compute community detection over the VIG to get a partition of the variables. Then, each clause is assigned to the most frequent community among its variables (randomly assigned in case of ties). This generates a partition of clauses with a lower cost than running the Louvain algorithm on the CIG.

7.3.2 *Correlating*

The LBD [11] measure of a learnt clause is the number of decision levels represented by its variables (lower is better). In particular, clauses that have LBD value of two (called *glue*

clause) are very important because they allow the merging of two decision levels. This measure is now used in almost all modern competitive solvers.

As pointed out by [92], there exists a strong correlation between the LBD measure, used to quantify the quality of a learnt clause, and the number of communities represented in this clause: a clause with a low LBD has variables dispatched in a few communities. This correlation is very important and permits a better theoretical understanding of the practical efficiency of the state-of-the-art LBD measure.

7.3.3 *Exploiting*

Based on the previous observation, the idea proposed in [4] is to generate clauses involved in a few communities of variables (less or equal to three in practice) meaning they likely have a small LBD value. The proposed `modprep` preprocessor (see Section 3.6) derives “good” clauses to enrich the original formula. It first computes the community of clauses. Afterwards each adjacent pair of clauses communities (*i.e.,* with a non-empty intersection of their variables) is solved independently, and learnt clauses are kept to enrich the initial formula. Finally, the enriched formula is given to a SAT solver.

7.4 Generating “Good” Clauses

The preprocessor proposed in [4] generates “good” clauses. Our goal here is to go further and generate more of these “good” clauses to accelerate SAT solving.

In practice the number of clauses generated by such a preprocessing is small on many instances. Indeed, the generated sub-formulas are often quickly reported SAT thus leading to few conflicts and generating few clauses. This phenomenon is due to the fact that branching only on some central variables of these formulas satisfies them since clauses in communities share many variables (by definition).

7.4.1 *Proposed Method*

To overcome this quick solving of the generated sub-formulas, we propose to shuffle the clauses. We create a shuffled partition of clauses from the communities of clauses by using the round-robin principle. As all the generated parts have a non-empty intersection, we propose to use a solving algorithm mechanism inspired by [50] instead of solving each pair.

Algorithm 8 presents the procedure we used as preprocessing. The original formula φ is decomposed in multiple sub-formulas (ψ_i) according to the strategy defined by the `decompose` function (Line 2). Then a merging formula (Γ) is initialised to the empty set (Line 3). At each turn of the loop, Γ is solved. If Γ is UNSAT then the original formula is reported UNSAT

(Line 7). Otherwise each of the sub-formulas (ψ_i) is solved using the model of Γ as assumptions (Line 11). If the sub-formula (ψ_i) is proven UNSAT the derived clauses is added to Γ (Line 13). If all the sub-formulas (ψ_i) are reported SAT then the original formula is SAT and the model is the one computed for Γ (Line 15).

```
1  function solveDecompose(φ: a CNF formula)
       /* returns ⊤ if φ is SAT else ⊥                                  */
2      ψ_1, ..., ψ_k ← decompose(φ)
3      Γ ← {}
4      is_sat ← ⊤
5      while is_sat = ⊥ do
6          if solve(Γ) = ⊥ then
7              return ⊥
8          Let m be a model of Γ
9          is_sat ← ⊤
10         foreach i ∈ [[1, k]] do
11             if solve(ψ_i ∧ m) = ⊥ then
12                 Let ω be the final conflict analysis clause derived from ψ_i ∧ m
13                 Γ ← Γ ∪ {ω}
14                 is_sat ← ⊥
15     return ⊤
```

Algorithm 8: Solving procedure through formula decomposition

Note that this procedure is complete and can solve the original formula. Our goal is to use this procedure to derive "good" clauses produced by the final conflict analysis of the algorithm (Line 13).

7.4.2 *Evaluation of "Good" Clauses Production*

We use as decomposition two techniques: the round-robin method presented above and a random decomposition. The random decomposition has been parametrised to generate a partition with the same size as the one created by using the round robin one. The algorithm while parametrised to use the round robin or the random as decomposition is respectively named `RR` and `RAND`.

The following experiments have been conducted on a machine with 40 processors Intel Xeon CPU E7 - 2860 @ 2.27GHz, and 500 GB of memory. We used a subset of the benchmark from the SAT Competition 2018. This subset contains 357 of the 400 original instances, the one for which Louvain method has finished in less than 500 seconds. We have run the three preprocessors namely: `modprep`, `RR`, and `RAND`. For all the preprocessors the underlying sequential solver used is `MapleCOMSPS` [71]. The `modprep` preprocessor has been launched with no timeout, and `RR` and `RAND` with a timeout of 500 seconds.

Note that modprep (that has been launched without timeout) finishes 6 instances in more than 500 seconds.

modprep produces clauses that have one, two or three communities of variables represented. These clauses can be qualified as "good" clauses as they have a good impact on the efficiency of some solvers according to [4]. Hence, "bad" clauses are the other ones (*i.e.,* with more than three communities of variables).

Figure 7.2 reports the distribution of "good" clauses learnt for each instance by each of the three preprocessors. RR and RAND generates more clauses in average (66981 and 72372, respectively) compared to modprep (50094). Moreover, an interesting thing is that there are fewer instances where no clauses are generated for RR and RAND compared to modprep. And the median of clauses produced by RAND and RR (50945 and 55859, respectively) is clearly greater than the median of "good" clauses produced by modprep (219).

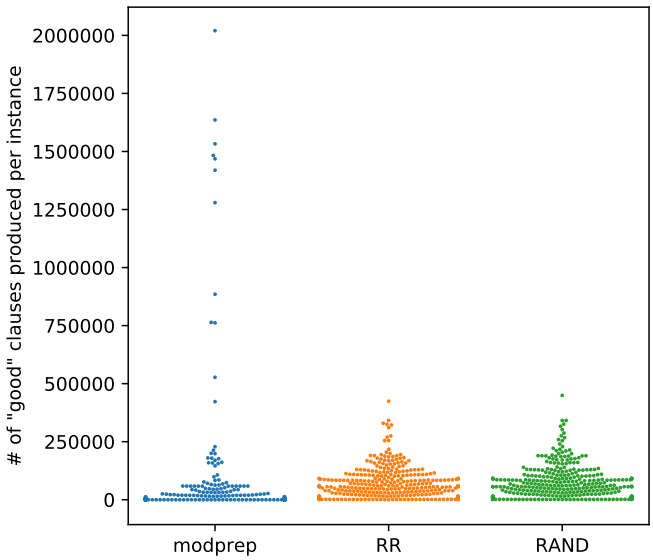


Figure 7.2: Swarmplot representing the distribution of "good" clauses generated by the different preprocessors

7.5 "Good" Preprocessing for Sequential SAT Solving

In this section we analyse the impact of "good" clauses generated with the different preprocessors on the sequential solver MapleCOMSPS [71]. For each of the 357 instances we used initially, we have run MapleCOMSPS on the original instance and the different instances that have been enriched with "good" clauses generated by the three presented preprocessors. We call the original benchmark and the benchmarks generated by adding "good" clauses from modprep, RR, and RAND, respectively *SC2018*, *SC2018-modprep*, *SC2018-RR*, and *SC2018-RAND*.

All the experiments were executed on a machine with 40 processors Intel Xeon CPU E7 - 2860 @ 2.27GHz, and 500 GB of memory. The used timeout is 5000 seconds.

Benchmark	**PAR-2**	**ALL (357)**	**SAT**	**UNSAT**
SC2018	2027401.91	169	108	61
SC2018-modprep	**2023288.51**	**170**	**109**	61
SC2018-RR	2025088.83	168	106	**62**
SC2018-RAND	2052433.58	166	105	61

Table 7.3: Results of `MapleCOMSPS` on the different generated benchmarks

Results of these experiences are summarised in Table 7.6. The different columns represent respectively: the PAR-2 score, the total number of solved instances; the number of SAT solved instances, and the number of UNSAT solved instances. `MapleCOMSPS` has a better PAR-2 on both benchmarks *SC2018-modprep* and *SC2018-RR* compared to the original benchmark *SC2018*. This seems to confirm the fact that "good" clauses generated by the preprocessors `modprep` and `RR` help to speed up the solving process.

v.s. / Benchmark	*SC2018*	*SC2018-RR*	*SC2018-RAND*	*SC2018-modprep*
SC2018		+0h58 (154)	+0h12 (150)	**-2h25 (157)**
SC2018-RR	**-0h58 (154)**		**-1h49 (152)**	**-3h25 (152)**
SC2018-RAND	**-0h12 (150)**	+1h49 (152)		**-2h33 (151)**
SC2018-modprep	+2h25 (157)	+3h25 (152)	+2h33 (151)	

Table 7.4: Differences in terms of CTI of `MapleCOMSPS` on the different pairs of preprocessed benchmarks

Table 7.4 presents the difference in terms of CTI of `MapleCOMSPS` on the different pairs of benchmarks. Each line gives the difference between the CTI of `MapleCOMSPS` for one benchmark *w.r.t.* the other benchmarks. We can notice that `MapleCOMSPS` is always faster (in terms of CTI) when solving the *SC2018-RR* benchmark. Regarding the 152 instances solved by `MapleCOMSPS` for the *SC2018-RR* and *SC2018-modprep* the saved time is 3h25. Moreover, we can see that `MapleCOMSPS` is slower when considering the CTI for the *SC2018-modprep* *w.r.t.* the other benchmarks. These results are really encouraging for our approach, and the generated "good" clauses seems to have a positive impact on the solving process of `MapleCOMSPS`.

`MapleCOMSPS` is slower on the *SC2018-RAND* benchmark compared the two others in terms of PAR-2. But it `MapleCOMSPS` is faster considering only the CTI of instances from the *SC2018-RAND* *w.r.t.* to the *SC2018* and *SC2018-modprep* benchmarks. For every measure `MapleCOMSPS` performs better on *SC2018-RR* compared to *SC2018-RAND*. The "good" clauses generated by the `RAND` preprocessor do not seem to be as "good" as the one produced by `RR`. One question arises: is the definition of "good" clause too inclusive? Authors

of [92] highlight a correlation between LBD and number of communities. **The reverse does not always seem to be true.**

7.6 “Good” Inprocessing for Parallel SAT Solving

This section presents the use of the RR algorithm as inprocessor in the context of parallel SAT solving. RR is interesting in the parallel context as it is a complete procedure that will compute good clauses until the end of of global solving. First, we explain how we have incorporated this inprocessing in the P-Syrup solver. Afterwards, some results are presented to show how this inprocessing technique acts on the efficiency of the P-Syrup solver.

7.6.1 *Implementation*

This section explains how we have integrated RR as inprocessing in the P-Syrup solver (see Section 5.2.1).

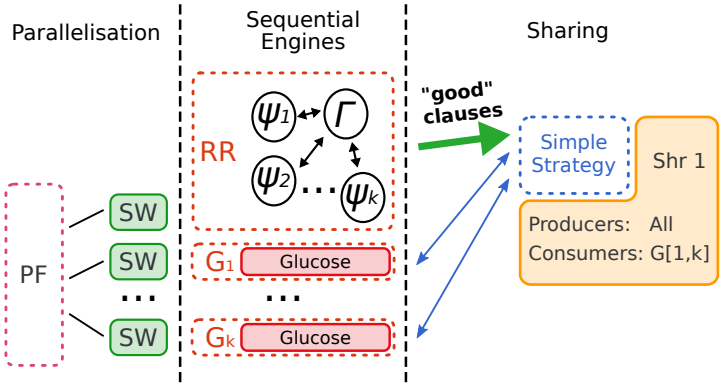


Figure 7.5: Architecture of P-Syrup-RR

Figure 7.5 depicts the general architecture of the P-Syrup-RR solver. First, we have created a SolverInterface that encapsulates the RR mechanism. The RR solver relies on the use of instances of the MapleAdapter. This solver is part of the Portfolio strategy with $N - 1$ GlucoseAdapter. The native diversification used in P-Syrup is applied on these $N - 1$ solvers. The used sharing strategy is the same as the one used in P-Syrup: SimpleSharing (see Section 5.2.1). The RR solver is producer of clauses (*i.e.,* the one that are qualified as “good”), and is not a consumer.

7.6.2 *Experimental Evaluation*

All experiments were executed on a multi-core machine with 12 physical cores (2 x Intel Xeon E5645 @ 2.40 GHz), and 64 GB of memory. We used the complete benchmarks (*i.e.,* 400 instances) from the SAT Competition 2018 and a timeout of 5000 seconds.

Results of these experiences are summarised in Table 7.6. The different columns represent respectively: the PAR-2 score, the total number of solved instances, the number of UNSAT solved instances, and the number of SAT solved instances.

Solver	**PAR-2**	**ALL (400)**	**SAT**	**UNSAT**
P-Syrup	1941049.58	222	113	109
P-Syrup-RR	**1910708.30**	**226**	**117**	109

Table 7.6: Results of P-Syrup and P-Syrup-RR on the benchmark from the SAT Competition 2018

The version using RR as inprocessing is able to solve 4 more instances than the original P-Syrup. All these instances are SAT, this corroborates the fact that these "good" clauses have more impact on SAT instances as stated in [4]. Moreover, P-Syrup-RR has a better PAR-2 measure compared to the original P-Syrup solver (1910708.30 *vs.* 1941049.58). This showing the good impact of adding the RR inprocessing to produce "good" clauses during the whole solving.

7.7 CONCLUSION AND DISCUSSION

Clause sharing in parallel SAT solving is very important and warrant a particular focus. In practice not all the learnt clauses can be shared between the workers. In this chapter we explored strategies to quantify learnt clause quality. This quantification is very important to improve selection of the clauses that should be shared.

We proposed RR a preprocessing/inprocessing producing "good" clauses speeding up the solving of SAT instances. This work is based on an algorithm from the literature [50] and strengthens the work proposed in [4].

The efficiency of RR has been highlighted with preliminary results. In the sequential context RR has been used as preprocessing, in parallel it is used as inprocessing.

We plan to conduct more experiments, using different benchmarks and solvers, to corroborate the efficiency of RR.

Since SAT instances are huge, applying the Louvain method is sometimes expensive. Moreover, it is sometimes useless as solving the instance can be faster than applying community detection. Deciding heuristically if the method should be launched or not, regarding the instance or the graph, could help.

Finally, sharing should maybe adapt in order to prioritise the exchange of "good" clauses. These clauses could be exchanged in priority through the use of special queues. Also the quality of clauses produced by classical sequential solvers could be quantified using structural properties of the formula. These kinds of measures are global to all workers contrariwise to the LBD, for instance, that is local.

Chapter

8

Conclusion and Future Works

This thesis dealt with the parallelisation of SAT solving. One of the reasons for the success of SAT is the practical efficiency of SAT solvers. With the omnipresence of parallel environments (*e.g.,* many-core machines, cloud) SAT solving needs to enter a new era. Unfortunately, designing efficient parallel SAT solvers is a challenging task that we mainly simplified in the work presented in this thesis.

After a study of the literature (see Chapters 3 and 4) we have highlighted tree main challenges for improving parallel SAT solving: (C1) filling the gap between divide-and-conquer and portfolio strategies, (C2) improving clause sharing both theoretically and technically, and (C3) improving the tooling support for parallel SAT solving. The first two challenges (C1 and C2) clearly heavily rely on the solution of the last challenge (C3).

To tackle the challenge C3 we have proposed `Painless`, a modular, generic and efficient framework for parallel SAT solving (see Chapter 5). We claimed that its modularity and genericness allow the implementation of basic strategies, as well as new ones and their combination with minimal effort and concurrent programming skills. We have shown the genericness of `Painless` by the implementation of strategies present in state-of-the-art solvers: `Syrup`, `Treengeling`, and `HordeSat`. Second, we highlighted its modularity by reusing developed complements to derive, easily, new solvers that mix strategies. We have also shown that the instantiated solvers are as efficient as the original ones. Moreover, the efficiency of solvers generated with `Painless` has been assessed during the SAT Competition 2017 and 2018 where `P-MapleCOMSPS` has been ranked 3rd and 1st, respectively.

To cope with challenge C1, we have extended the original version of `Painless`, by adding a modular and efficient divide-and-conquer component on top of it (see Chapter 6). Our `Painless`-based implementation can be customised and adapt over six orthogonal mechanisms:

- (M1) technique to divide the search space;
- (M2) technique to choose the division variables;

- (M3) dynamic load balancing strategy;
- (M4) the sharing strategy;
- (M5) the workers' knowledge management;
- and (M6) the used underlying sequential solver.

We were then able to make a fair comparison between numerous strategies. Among the numerous solvers we have available, our original association between the flip-based heuristic and the clone policy outperforms other strategies and state-of-the-art `D&C` solvers (*i.e.,* `MapleAmpharos` and `Treengeling`).

From a technical point of view, C2 has been addressed by providing an efficient and modular sharing mechanism embedded in `Painless`. This mechanism is one of the main reasons for the `Painless`' efficiency. It is based on the used of lock free buffers and decentralised synchronisation. Considering the theoretical aspect of the challenge C2, Chapter 7 presented heuristics to exploit the modularity of SAT instances in order to derive "good" clauses. In practice this modularity is highlighted by computing the community structure on a graph representing the SAT instance. A novel technique inspired by the literature is proposed as preprocessor (inprocessor) to generate clauses accelerating solving in the sequential (parallel) context.

8.1 PERSPECTIVES

This section presents two main perspectives identified as interesting future works: (1) designing scalable SAT solvers, and (2) generalising the `Painless` framework.

8.1.1 *Towards Scalable Parallel SAT Solving*

The main issue for the scaleability of parallel SAT solvers is the fact that the more workers are involved in the solving the more clauses are exchanges. When increasing the number of workers the number of exchanged clauses increases exponentially. There are two axes to reduce exchanges: (1) limiting the number of clauses, and (2) limiting the number of workers involved in the sharing. Both can be done statically or dynamically.

In classical parallel SAT solvers, both axes are treated statically: all the workers send clauses to each other, and sharing is done using static heuristics. In order to scale, SAT solvers should incorporate dynamic selection.

For the first axis, a solution is to not allow all the workers to communicate together. To the best of our knowledge there exists only one work exploring this direction: in [64] authors proposed that each worker could select dynamically the ones that can send it clauses. A more general possibility is the organisation of the workers in *groups of sharing*. We can imagine different

heuristics to group workers based on (1) system observations (*e.g.,* workers collocated on the same NUMA node), or on (2) algorithmic observations (*e.g.,* workers exploring certain part of the search space). It is also possible to combine the different dimensions to group the workers.

Considering the second axis, there exist more contributions but many of them are static. Sharing is then limited to only very "good" clauses. To tackle this point one need to define strategies that should be more adaptive to the number of involved workers, like the AIMD algorithm [48], to control the flow of clauses. Even if this last work proposes an adaptive strategy, it relies on a fix threshold. Moreover, even if there exist quality metrics for learnt clauses (*e.g.,* LBD [11]), more global metrics (*i.e.,* that do not depend on the sequential context of a particular solver) should be defined.

Finally, nowadays parallel SAT solvers run a worker per available CPU, each worker having a copy of the problem and its own solving structures. A solution that could be explored is the use of a unique structure to manipulate different components embedded in sequential SAT solvers (*e.g.,* learnt clause database). This allowing a more systematic lazy sharing. The problem of this approach is the synchronisation required by such a shared data structure.

8.1.2 *Generalising the Modular Approach*

The creation of a modular and efficient framework for concurrent SAT solving (*i.e.,* `Painless`) has allowed us to compare multiple techniques and heuristics. Hence, we have been able to highlight some combination of heuristics that work well together. This principle can be extended to other environments and variants of the SAT problem.

Next step would be to design a version that could work on distributed environments. While designing a generic distributed level, we want to continue to have the genericness and efficiency at the machine level. A classical solution used in the context of distributed algorithm is the use of a hierarchy. Adding a hierarchical organisation into `Painless` should be done while preserving the modularity of both sharing and working. A starting point could be the use composition of `TopoSAT` [39] between machines and `Painless` between cores of a computer.

An interesting work would be to adapt a version for solving other problems such as #SAT [20, Chapter 20], MAX-SAT [20, Chapter 19], and SMT [20, Chapter 26]. To do so, a bibliographical study of the different practices and heuristics used in these fields should be done. Indeed, one wants to be able to implement every algorithm from the literature, but also new ones.

List of Figures

Bibliography

[1] Amd64 architecture programmer's manual volume 2: System programming, 2017. https://support.amd.com/TechDocs/24593.pdf.

[2] Fadi A Aloul, Igor L Markov, and Karem A Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference (DAC)*, pages 836–839. ACM, 2003.

[3] 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.

[4] 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.

[5] Alejandro Arbelaez and Philippe Codognet. A gpu implementation of parallel constraint-based local search. In *Processing of the 22nd International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 648–655. IEEE, 2014.

[6] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[7] 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.

[8] 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.

[9] 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.

[10] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. A distributed version of syrup. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 215–232. Springer, 2017.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, and Joost Vennekens. Parallel hybrid sat solving using opencl. In *Proceedings of the 24th Benelux Conference on Artificial Intelligence (BNAIC)*, pages 11–18. Springer, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] Wolfgang Blochinger. Towards robustness in parallel sat solving. In *Proceedings of the 11th International Parallel Computing Conference (PARCO)*, pages 301–308, 2005.

[23] 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.

[24] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[25] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

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

[27] 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.

[28] Carlos Filipe Costa. Parallelization of sat algorithms on gpus. Technical report, Technical report, INESC-ID, Technical University of Lisbon, 2013.

[29] 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.

[30] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

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

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

[33] 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.

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

[35] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for sat. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 104–122. Springer, 2016.

[36] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2011.

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

[38] 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.

[39] Thorsten Ehlers and Dirk Nowotka. Tuning parallel sat solvers. In *Pragmatics of SAT International Workshop (POS) at SAT*, 2018.

[40] 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.

[41] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

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

[43] Hironori Fujii and Noriyuki Fujimoto. Gpu acceleration of bcp procedure for sat algorithms. *IPSJ SIG Notes*, 8:1–6, 2012.

[44] François Galea and Bertrand Le Cun. Bob++: a framework for exact combinatorial optimization methods on parallel machines. In *Proceedings of the 5th International Conference High Performance Computing & Simulation (HPCS)*, pages 779–785, 2007.

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

[46] 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.

[47] 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.

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

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

[50] Youssef Hamadi, Joao Marques-Silva, and Christoph M Wintersteiger. Lazy decomposition for distributed decision procedures. In *Proceedings of the 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*. Electronic Proceedings in Theoretical Computer Science, 2011.

[51] Pierre Hansen, Nenad Mladenović, and Dionisio Perez-Britos. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4):335–350, 2001.

[52] Jin-Kao Hao and Raphaël Dorne. An empirical comparison of two evolutionary methods for satisfiability problems. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 450–455. IEEE, 1994.

[53] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 228–245. Springer, 2016.

[54] 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.

[55] 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.

[56] Antti EJ Hyvärinen, Tommi Junttila, and Ilkka Niemela. Incorporating clause learning in grid-based randomized sat solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):223–244, 2009.

[57] Antti EJ Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning search spaces of a randomized search. In *Proceedings of the 12th Conference of the Italian Association for Artificial Intelligence (AI*IA)*, pages 243–252. Springer, 2009.

[58] Antti EJ Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 214–227. Springer, 2012.

[59] Antti EJ Hyvärinen and Christoph M Wintersteiger. Approaches for multi-core propagation in clause learning satisfiability solvers. Technical Report MSR-TR-2012-47, Microsoft Research, 2012.

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

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

[62] 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.

[63] 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.

[64] Nadjib Lazaar, Youssef Hamadi, Said Jabbour, and Michèle Sebag. Cooperation control in parallel sat solving: a multi-armed bandit approach. Technical Report RR-8070, INRIA, 2012.

[65] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Kordon Fabrice. painless-maplecomsps. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, pages 26–27. Department of Computer Science, University of Helsinki, Finland, 2017.

[66] 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.

[67] 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.

[68] 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.

[69] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, pages 115–116, 1973.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Norbert Manthey. Towards improving the resource usage of sat-solvers. In *Pragmatics of SAT International Workshop (POS) at SAT*, 2010.

[77] Norbert Manthey. Coprocessor–a standalone sat preprocessor. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP)*, pages 297–304. Springer, 2011.

[78] Norbert Manthey. Parallel sat solving-using more cores. In *Pragmatics of SAT International Workshop (POS) at SAT*, 2011.

[79] Norbert Manthey. Coprocessor 2.0–a flexible cnf simplifier. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 436–441. Springer, 2012.

[80] 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.

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

[82] Deborah T Marr. Hyperthreading technology architecture and microarchitecture: a hyperhtext history. *Intel Technology Journal*, 6:1, 2002.

[83] 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.

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

[85] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in sat solving. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 99–114. Springer, 2018.

[86] Quirin Meyer, Fabian Schönfeld, Marc Stamminger, and Rolf Wanka. 3-sat on cuda: Towards a massively parallel sat solver. In *Proceedings of the 8th International Conference on High Performance Computing and Simulation (HPCS)*, pages 306–313. IEEE, 2010.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

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

[92] 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.

[93] Kei Ohmura and Kazunori Ueda. c-sat: A parallel sat solver for clusters. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 524–537. Springer, 2009.

[94] Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26(1):35–62, 1988.

[95] 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.

[96] 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.

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

[98] Mark Redekopp and Andreas Dandalis. A parallel pipelined sat solver for fpga's. In *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL)*, pages 462–468. Springer, 2000.

[99] Sven Schulz and Wolfgang Blochinger. Cooperate and compete! a hybrid solving strategy for task-parallel sat solving on peer-to-peer desktop grids. In *High Performance Computing and Simulation*, pages 314–323. IEEE, 2010.

[100] Harald Seltner. Extracting hardware circuits from cnf formulas, 2014. http://fmv.jku.at/master/Seltner-MastherThesis-2014.pdf.

[101] 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.

[102] Daniel Singer and Anthony Monnet. Jack-sat: A new parallel scheme to solve the satisfiability problem (sat) based on join-and-check. In *Processing of the 7th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 249–258. Springer, 2007.

[103] Iouliia Skliarova and Antonio de Brito Ferrari. A software/reconfigurable hardware sat solver. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4):408–419, 2004.

[104] 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.

[105] 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.

[106] 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.

[107] Mate Soos. Enhanced gaussian elimination in dpll-based sat solvers. In *Pragmatics of SAT International Workshop (POS) at SAT*, pages 2–14, 2010.

[108] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[109] Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. A collaborative approach for multi-threaded sat solving. *International Journal of Parallel Programming*, 37(3):324–342, 2009.

[110] 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.

[111] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, and Stefan Porschen. pfoliouzk: Solver description. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, page 45. Department of Computer Science, University of Helsinki, Finland, 2012.

[112] 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.

[113] 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.

[114] Peixin Zhong, Margaret Martonosi, and Pranav Ashar. Fpga-based sat solver architecture with near-zero synthesis and layout overhead. *IEE Proceedings-Computers and Digital Techniques*, 147(3):135–141, 2000.