SAT en Parallèle

THÈSE

présentée et soutenue publiquement le 12 décembre 2017

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Nicolas SZCZEPANSKI

Composition du jury

Président :	Chu-Min LI	Université de Picardie
Rapporteurs :	Gilles DEQUEN	Université de Picardie
	Michaël KRAJECKI	Université de Reims
Examinatrice :	Thi-Bich-Hanh DAO	Université d'Orléans
Co-Encadrants :	Jean-Marie LAGNIEZ	Université d'Artois
	Sébastien TABARY	Université d'Artois
Directeur de Thèse :	Gilles AUDEMARD	Université d'Artois

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.univ-artois.fr

Mise en page avec memcril (B. Mazure, CRIL) et thloria (D. Roegel, LORIA).

Remerciements

Les résultats de cette thèse sont ceux de toute une équipe.

D'abord et avant tout, je tiens à remercier mon directeur de thèse et mes encadrants : Gilles Audemard, Jean-Marie Lagniez et Sébastien Tabary. Ils m'ont apporté l'opportunité de travailler sur un sujet de recherche excitant. Plus encore, ils m'ont accompagné tout au long de ma thèse, sans jamais m'abandonner. Cette collaboration m'a permis de mieux comprendre divers aspects du sujet qui était le mien.

Je tiens aussi à remercier Gilles Dequen et Michaël Krajecki (rapporteurs) puis, Thi-Bich-Hanh Dao et Chu-Min Li (examinateurs) d'avoir accepté de rapporter et d'examiner cette thèse. Leurs remarques seront très utiles à la finalisation du manuscrit.

Je souhaite exprimer également ma reconnaissance à tous les membres du laboratoire.

D'abord, je remercie notre directeur, Éric Grégoire, de m'avoir accueilli au sein du CRIL. Je tiens à remercier tout particulièrement François Chevallier, pour le temps qu'il a consacré à la configuration du cluster. Je remercie aussi le personnel administratif : Virginie Delahaye, Frédéric Renard et Sandrine Saitzek, qui ont toujours été là pour m'aider (réservation, organisation de pots, ...). Sans oublier ceux qui m'ont conseillé sur divers points de recherche : Pierre Marquis, Bertrand Mazure et Olivier Roussel.

Merci aux autres thésards auprès de qui j'ai trouvé une stimulation et une connivence motivantes. Ceux avec qui tout a commencé, en Master : Jérôme Delobelle, Clément Lecat et Amélie Levray. Ceux qui ont rejoint le laboratoire pendant la préparation de ma thèse : Gaël Glorian et Valentin Montmirail, avec qui j'ai passé de bons moments (le Harem). Ainsi que les anciens : Zied Bouraoui, Thomas Caridroit et Emmanuel Lonca.

Ces remerciements seraient incomplets si je n'en adressais pas à ma famille. Je remercie mes deux grands-mères Josiane et Georgette, de m'avoir encouragé. Je remercie mon frère, Antonin et mon père, Patrice pour leur soutien et leur intérêt pour mes travaux. Je remercie le parrain de ma fille, Nicolas, qui a toujours cru en moi.

Mon remerciement le plus fort sera pour Élodie. Merci de m'avoir soutenu tout au long de cette thèse. En plus de supporter mes quelques mauvaises humeurs quotidiennes, tu t'es occupée de notre fille, pendant mes absences et plus particulièrement à la fin de cette thèse. Merci de vivre à mes côtés et de m'avoir permis de réalisé ma thèse, mais aussi, de d'associer à moi pour fonder une famille.

Je clos ces remerciements en dédiant cette thèse à ma fille, Théa, qui me comble de bonheur.

À ma fille, Théa.

Table des matières

Remerciements **i**

Liste des tableaux **xi**

Table des figures **xiii**

Introduction générale **1**

État de l'art

Chapitre 1 Le problème SAT **9**

1.1 Logique propositionnelle 10

1.1.1 Syntaxe 10

1.1.2 Sémantique 11

1.1.3 Formes normales 13

1.2 Théorie de la complexité 14

1.3 Problème SAT 15

1.3.1 Encodages 15

1.3.2 Fragments polynomiaux 16

1.4 Applications 17

1.4.1 Exemples 18

1.5 Conclusion 21

Chapitre 2 Résolution séquentielle du problème SAT 23

2.1 Approches complètes ou incomplètes 23

2.1.1 Approches incomplètes 24

2.1.2 Approches complètes 25

2.1.3 Comparaison 25

2.2 Genèse des solveurs SAT 26

2.2.1 Résolution 26

2.2.2 L'algorithme Davis - Putnam 28

2.2.3 L'algorithme de Davis - Logemann - Loveland 29

2.3 Les solveurs SAT 34

2.3.1 Apprentissage 34

2.3.2 L'algorithme CDCL 38

2.3.3 Choix de variable et choix de polarité 40

2.3.4 Structure de données paresseuses 45

2.3.5 Politiques de suppression des clauses apprises 48

2.3.6 Stratégies de redémarrage 51

2.3.7 Prétraitement 53

2.3.8 *Inprocessing* 57

2.4 Conclusion 57

Chapitre 3 Le parallélisme 59

3.1 Introduction 60

3.2 Architectures 62

3.2.1 Architecture de Von Neumann 62

3.2.2 Registre et mémoire cache 63

3.2.3 Pipeline 64

3.2.4 Architectures parallèles 66

3.2.5 Processeur vectoriel 67

3.2.6 Processeur multi-cœur 69

3.3 Architectures de la mémoire en parallèle 70

3.3.1 Mémoire partagée . . . 70
3.3.2 Mémoire distribuée . . . 72
3.3.3 Hybridation mémoire partagée/distribuée . . . 72
3.4 Évolution des architectures . . . 73
3.4.1 Processeur many-cœur . . . 73
3.4.2 Super-ordinateur . . . 73
3.5 Principes de la parallélisation . . . 74
3.5.1 Notions . . . 74
3.5.2 Limites . . . 75
3.5.3 Accélérations théoriques . . . 79
3.6 Modèles parallèles . . . 82
3.6.1 Sortes de parallélismes . . . 83
3.6.2 Modèles de programmation parallèle . . . 84
3.7 Conclusion . . . 87

Chapitre 4 Résolution parallèle du problème SAT **89**
4.1 Approche « *portfolio* » . . . 90
4.1.1 MANYSAT : Diversification *Versus* Intensification . . . 91
4.1.2 PLINGELING : De nombreuses techniques *inprocessing* . . . 93
4.1.3 DP^{2}LL : Un solveur déterministe . . . 94
4.1.4 PPFOLIO : Une approche simple . . . 95
4.1.5 BESS : Les bandits manchots . . . 96
4.1.6 PENELOPE : Les clauses gelées . . . 96
4.1.7 MINIRED et GLUCORED : Un raffinement des clauses . . . 97
4.1.8 SYRUP : Un échange paresseux . . . 98
4.1.9 CBPENELOPE et PARACIRMINISAT : Exploiter les communautés . . . 99
4.1.10 TOPOSAT : Les topologies du partage des clauses . . . 100
4.1.11 HORDESAT : Hybridation et diversification . . . 102
4.2 Approches « diviser pour mieux régner » . . . 103
4.2.1 PSATO : La méthode « chemin de guidage » . . . 104

4.2.2 MTSS : *Thread* riche et *threads* pauvres 106
4.2.3 C-SAT : Plusieurs divisions en concurrence 107
4.2.4 SATCIETY : Du *peer-to-peer* (P2P) 108
4.2.5 PART-TREE-LEARNING : L'échange des clauses 108
4.2.6 CUBEANDCONQUER : Une décomposition statique 109
4.2.7 DOLIUS : Une API simple et clair 112
4.3 Conclusion 113

Contributions **115**

Chapitre 5 De la décomposition statique à celle dynamique **117**
5.1 Décomposition via les bandits manchots : UCTSAT 117
5.1.1 Utilisation d'UCT pour générer des cubes 118
5.1.2 Algorithme 119
5.1.3 Descente 121
5.1.4 Sélection 125
5.1.5 Simulation 127
5.1.6 Remontée 127
5.1.7 Récupération des cubes 127
5.2 Décomposition via CUBEANDCONQUER : SWARMSAT 128
5.2.1 Les différents processus 128
5.2.2 Communication 129
5.2.3 Mode de fonctionnement 129
5.3 Expérimentations 130
5.3.1 UCTSAT 130
5.3.2 SWARMSAT 131
5.4 Conclusion 132

Chapitre 6 AMPHAROS : Un solveur « diviser pour mieux régner » distribué **135**

6.1 La gestion de l'arbre 136

6.1.1 Initialisation 137

6.1.2 Transmission 138

6.1.3 L'extension 139

6.1.4 L'élagage 140

6.2 L'échange des clauses 141

6.2.1 Le partage classique des clauses apprises 141

6.2.2 Les littéraux unitaires sous hypothèses 142

6.3 Intensification vs Diversification 144

6.3.1 Évaluation du degré de redondance 144

6.4 Évaluation expérimentale 146

6.4.1 Gestion des communications 146

6.4.2 Configuration 146

6.4.3 Résultats 147

6.5 Conclusion 151

6.5.1 MAPLEAMPHAROS : Le ratio de propagations par décision 153

Chapitre 7 D-SYRUP : Un solveur « *portfolio* » distribué **155**

7.1 Étude expérimentale de SYRUP 156

7.1.1 Instances résolues 156

7.1.2 Extensibilité 157

7.2 L'architecture distribuée 160

7.2.1 AMPHAROS 160

7.2.2 Interblocages 161

7.2.3 Les cycles de communications 161

7.2.4 Phase de recherche et phase de communications 162

7.2.5 Objectifs 162

7.3 Le modèle de programmation pur par passage de messages 162

7.3.1 Désavantages 163

7.3.2 Avantages 164
7.3.3 Utilisations 164
7.3.4 Expérimentations 164
7.4 Le modèle de programmation partiellement hybride 166
7.4.1 Avantages et désavantages 166
7.4.2 Utilisation 167
7.4.3 Implémentation 168
7.4.4 Expérimentations 168
7.5 Le modèle de programmation complètement hybride 169
7.5.1 Méthode 170
7.5.2 Le problème des interblocages sur le réseau 171
7.5.3 Expérimentations 171
7.6 Expérimentations 172
7.6.1 Comparaison des différents modèles de programmation 172
7.6.2 Évaluation de D-SYRUP 174
7.7 Conclusion 177

Conclusion et perspectives **181**

Index **185**

Bibliographie **189**

Liste des tableaux

1.1 Sémantique usuelle des opérateurs logiques 11

2.1 Table de vérité d'une formule 25
2.2 Résultats de différentes approches SAT (complètes, incomplètes et hybrides) 25
2.3 Évolution de la propagation avec une structure de données paresseuse (2-*watch*) 48

3.1 Hiérarchie de la mémoire 64
3.2 Instruction dans un pipeline de type RISC classique 65
3.3 Deux instructions exécutées sur un processeur sans pipeline 66
3.4 Deux instructions exécutées sur un processeur possédant un pipeline 66
3.5 La taxonomie de Flynn 67
3.6 Classification basée sur la granularité 83
3.7 Passage de messages avec interblocage (*deadlock*) 87
3.8 Passage de messages sans interblocage (*deadlock*) 87
3.9 Passage de messages dans un environnement *multi-threaded* 87

5.1 Information sur la création des cubes par UCTSAT 130
5.2 Résultats de UCTSAT sur 7 instances 130
5.3 Résultats des différentes versions de SWARMSAT contre DOLIUS 131

6.1 Évaluation des composants du solveur AMPHAROS 147
6.2 AMPHAROS contre les solveurs de l'état de l'art 149
6.3 Étude de la valeur $rscm$ d'AMPHAROS 151

7.1 Résultats de SYRUP sur 1 à 32 cœurs 157
7.2 Modèle de programmation pur par passage de messages (expérimentation) 165
7.3 Modèle de programmation partiellement hybride (expérimentation) 169
7.4 Modèle de programmation complètement hybride (expérimentation) 172
7.5 Résultats des trois différents modèles de programmation en nombre d'instances résolues. 173
7.6 Résultats de GLUCOSE, SYRUP, D-SYRUP, HORDESAT et TOPOSAT 175

Table des figures

1 Évolution des solveurs SAT séquentiels au fil de ces dernière années (*cactus plot*) 2

1.1 Modélisation d'un compteur à deux bits 18
1.2 Problème de la coloration d'un graphe 19
1.3 Problème d'un carré latin 20
1.4 Problème des triplets pythagoriciens booléens 20

2.1 Résolutions et Arbre de réfutation 27
2.2 Algorithme DP 28
2.3 Algorithme DPLL 32
2.4 Graphe d'implications 36

3.1 Évolution du nombre de transistors 61
3.2 Évolution de la finesse de gravure 61
3.3 Évolution de la fréquence d'horloge 62
3.4 Évolution du nombre de cœurs 62
3.5 Architecture de Von Neumann 63
3.6 Architecture des processeurs vectoriels 68
3.7 Architecture des processeurs multi-cœurs 69
3.8 Mono-cœur *Versus* Multi-cœurs 70
3.9 Architecture de la mémoire partagée UMA 71
3.10 Architecture de la mémoire partagée NUMA 71
3.11 Architecture d'un Xeon Phi 74
3.12 Accélérations (*speedup)* 76
3.13 Algorithme séquentiel de neuf tâches 76
3.14 Algorithme parallèle avec une accélération linéaire 77
3.15 Problème d'équilibrage de charge 77
3.16 Diminution d'un déséquilibre des charges 78
3.17 Gestion dynamique des tâches 78
3.18 Problème des dépendances des tâches 79
3.19 Ralentissement dû aux communications 79
3.20 Lois d'Amdahl et de Gustafson 82
3.21 Ensemble de triangles non raffinés 84
3.22 Ensemble de triangles raffinés 84

4.1 Approche « *portfolio* » . . . 90
4.2 Architecture du solveur MANYSAT . . . 92
4.3 Architecture des solveurs MINIRED et GLUCORED . . . 97
4.4 Architecture du solveur *portfolio* SYRUP . . . 99
4.5 Communautés dans CBPENELOPE et PARACIRMINISAT . . . 100
4.6 Topologie à 4 voisin dans TOPOSAT . . . 101
4.7 Approche « diviser pour mieux régner » . . . 103
4.8 La méthode « chemin de guidage » avant un rééquilibrage . . . 105
4.9 La méthode « chemin de guidage » après un rééquilibrage . . . 106
4.10 Architecture du solveur C-SAT . . . 107
4.11 Division dans le solveur CUBEANDCONQUER . . . 109

5.1 Étapes de l'algorithme UCT . . . 120
5.2 Structure des nœuds dans UCTSAT . . . 120
5.3 La descente dans UCTSAT . . . 122
5.4 Calcul des valeurs UCB dans UCTSAT . . . 123
5.5 Conflit ou littéral déjà affecté à une valeur opposée lors de la descente dans UCTSAT . . . 125
5.6 Littéral déjà affecté lors de la descente dans UCTSAT . . . 125
5.7 La sélection dans UCTSAT . . . 126
5.8 La remontée dans UCTSAT . . . 128
5.9 Résultats des différentes versions de SWARMSAT contre DOLIUS (*cactus plot*) . . . 132

6.1 Architecture globale d'AMPHAROS . . . 136
6.2 Initialisation d'AMPHAROS . . . 137
6.3 Initialisation de l'arbre d'AMPHAROS . . . 137
6.4 Transmission d'un sous-problème dans AMPHAROS . . . 138
6.5 Extension dans AMPHAROS . . . 140
6.6 Élagage dans AMPHAROS . . . 140
6.7 Littéraux unitaires sous hypothèse dans AMPHAROS . . . 142
6.8 Évaluation des composants du solveur AMPHAROS (*cactus plot*) . . . 147
6.9 Extensibilité d'AMPHAROS (*cactus plot*) . . . 148
6.10 AMPHAROS contre les solveurs de l'état de l'art (*cactus plot*) . . . 150
6.11 Étude de la valeur $rscm$ d'AMPHAROS (*cactus plot*) . . . 151
6.12 Étude de la valeur $rscm$ d'AMPHAROS (*scatter plot*) . . . 152

7.1 Résultats du solveur parallèle SYRUP (*cactus plot*) . . . 157
7.2 Extensibilité de SYRUP de 1 à 32 cœurs (instances résolues séquentiellement) . . . 158

7.3 Extensibilité de SYRUP de 1 à 32 cœurs (toutes les instances) 159
7.4 Le modèle de programmation pur par passage de messages 163
7.5 Le modèle de programmation partiellement hybride 167
7.6 Modèle de programmation complètement hybride 170
7.7 Résultats expérimentaux des trois différents modèles de programmation (*cactus plot*) . . 173
7.8 Modèles complètement et partiellement hybrides (*scatter plot*) (SAT et UNSAT) 174
7.9 Modèles complétement et partiellement hybrides (*scatter plot*) (par famille) 175
7.10 Résultats de GLUCOSE, SYRUP, D-SYRUP, HORDESAT et TOPOSAT (*cactus plot*) . . . 176
7.11 HORDESAT *Versus* D-SYRUP (256 cœurs) (*scatter plot*) 177
7.12 TOPOSAT *Versus* D-SYRUP (256 cœurs) (*scatter plot*) 177
7.13 D-SYRUP (128 cœurs) *Versus* D-SYRUP (256 cœurs) (*scatter plot*) 178

Introduction générale

De nos jours, les ordinateurs sont de plus en plus exploités afin de résoudre de nombreux problèmes réels émanant de différents domaines. Destiné à cette tâche, le problème SAT (pour problème de SATisfaisabilité booléenne) s'appuie sur une modélisation en logique propositionnel des applications pour les résoudre. L'une des raisons du succès de SAT est la mise à disposition pour l'utilisateur de solveurs efficaces. En effet, les résultats de ces derniers sont très impressionnants et permettent la modélisation d'un grand nombre de problèmes réels (pouvant contenir des millions de variables et de clauses). Parmi les problèmes qui ont fait le succès de SAT, nous pouvons citer la vérification de modèles bornés (Biere *et al.* 1999), la planification (Kautz et Selman 1996), la résolution de problèmes combinatoires tel que les quasigroupes (Bennett et Zhang 2004a) et les nombres de Ramsey et Van der Waerden (Herwig *et al.* 2007). Plus récemment, l'utilisation du paradigme SAT a permis de résoudre la conjecture des triplets pythagoriciens (Heule *et al.* 2016). En plus, le problème SAT est pilier de la théorie de la complexité. En effet, c'est le premier problème a avoir été prouvé NP-complet (Cook 1971). De ce fait, de nombreux autres problèmes (extraction de MUS (Marques-Silva et Lynce 2011), logique modale (Lagniez *et al.* 2017), ...) sont souvent résolus au moyen d'une réduction vers SAT plutôt qu'avec des démonstrateurs ad-hoc.

Longtemps étudié dans un contexte séquentiel, le problème SAT a fait éclore des algorithmes efficaces et puissants. Ceci s'explique à la fois par l'amélioration des solveurs SAT basés sur l'algorithme CDCL (apprentissage, VSIDS, *watched literal*, ...) et l'accroissement de la fréquence des processeurs. En effet, nous pouvons observer dans la figure 1 une amélioration constante des solveurs SAT séquentiels au cours de ces deux dernières décennies. Cela est dû, notamment, à l'émulation apportée par les nombreuses applications et compétitions liées à SAT. Ainsi, grâce à cette émulation, chaque année de nombreuses améliorations sont apportées aux solveurs SAT. Ainsi, en 2017, la technique *inprocessing* appelée *Learnt Clause Minimization* s'est vue être particulièrement efficace (Luo *et al.* 2017).

La parallélisation des algorithmes de résolution pour SAT constitue un autre axe d'amélioration. Ceci est renforcé par le fait qu'à l'heure actuelle la puissance de calcul d'un ordinateur ne se traduit plus par une augmentation des fréquences du microprocesseur, mais par l'augmentation du nombre de cœurs de calcul au sein de celui-ci. Par conséquent, afin d'utiliser tout le potentiel des ordinateurs, le parallélisme d'un solveur SAT est aujourd'hui une obligation. Qui plus est, grâce à l'essor du *cloud computing*, il est désormais possible de solliciter un nombre conséquent de machines virtuelles pouvant être disponibles en quelques secondes.

Dans la résolution parallèle du problème SAT, les solveurs peuvent être divisés en deux catégories. Premièrement, les solveurs de type *portfolio* lancent, sur la formule originale, différentes heuristiques/stratégies en concurrence (Audemard *et al.* 2012, Balyo *et al.* 2015, Biere 2014, Hamadi *et al.* 2009a;c, Roussel 2011). Pendant la résolution, les processus échangent des informations (généralement sous la forme de clauses apprises) afin de s'entraider (Audemard *et al.* 2012, Audemard et Simon 2014, Hamadi *et al.* 2009a;c). Deuxièmement, les solveurs basés sur le paradigme « diviser pour mieux régner » découpent l'espace de recherche en plusieurs sous-espaces afin de les distribuer à des solveurs SAT, communément nommés *workers*. En général, à chaque fois qu'un solveur a fini de résoudre son sous-problème (tandis que les autres travaillent encore), une stratégie d'équilibrage est appliquée afin de transférer dynamiquement un nouveau sous-espace à ce *worker* inactif (Chrabakh et Wolski 2007, Chu *et al.* 2008). Les sous-espaces peuvent être définis en utilisant le concept du chemin de guidage (*guiding*

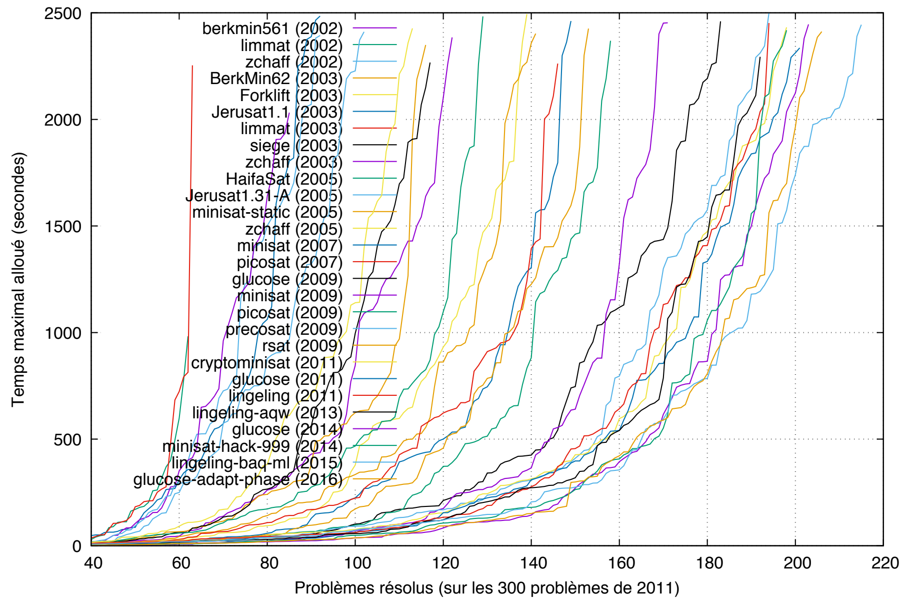


FIGURE 1 – Évolution des solveurs SAT séquentiels au fil de ces dernière années (*cactus plot*) sur les 300 instances de la compétition SAT 2011 (*main track*). Pour chaque solveur, les instances sont triées dans l'ordre croissant en fonction de leurs temps de résolutions (source : Laurent Simon).

path (Zhang *et al.* 1996)), généré statiquement (avant la recherche (Heule *et al.* 2012, Semenov et Zaikin 2015)), ou dynamiquement (pendant la recherche (Audemard *et al.* 2014b, Hyvärinen *et al.* 2010a, van der Tak *et al.* 2014)). Comme dans les solveurs de type *portfolio*, les clauses apprises peuvent aussi être partagées (Feldman *et al.* 2005).

Malheureusement, l'adaptation des algorithmes SAT vers le parallélisme se complexifie avec ses nombreuses améliorations séquentielles, mais aussi, à cause des nouvelles architectures mises en place par les constructeurs. Plus précisément, un nouveau type d'architecture appelé *many-cores* demande aux programmeurs de changer leurs manières de coder. Cela est déjà arrivé dans le passé puisque un programmeur en parallélisme doit aujourd'hui posséder des notions à la fois liées au *multi-cores* et au passage de messages entre plusieurs machines (*socket*, MPI, ...). En plus de cela, les solveurs SAT parallèles ne passent pas à l'échelle. Précisément, les gains dûs au parallélisme des solveurs SAT s'amenuisent à partir d'un certain nombre d'unités de calcul. Notamment, lors de la SAT compétition 2017, le gagnant du *Parallel Track* a choisi d'utiliser seulement la moitié des cœurs d'une machine, c'est-à-dire, 24 des 48 cœurs disponibles. Même si cette compétition utilisait 48 cœurs de calcul, 24 d'entre eux étaient *hyper-threaded*. En pratique, grâce aux résultats de cette compétition, nous pouvons nous apercevoir que l'*hyper-threading* n'est pas toujours efficace avec les solveurs SAT parallèles.

Le principale désavantage dans le *multi-cores* est son nombre limité d'unité de calcul. Pourtant, une manière pour un solveur SAT d'exploiter le potentiel du parallélisme est l'utilisation de plusieurs ordinateurs. Alors qu'une seule machine à un nombre limité d'unités de calcul, les *clusters* de calcul peuvent en apporter beaucoup plus et s'étendre en théorie à l'infini. Néanmoins, un solveur *mutli-thread* utilisant uniquement une librairies gérant la mémoire partagée ne peut pas exploiter correctement des techniques propres à SAT (partages des clauses, décomposition du problème, ...) sur plusieurs machines. Afin de pallier ce problème, il est alors nécessaire de basculer vers les architectures distribuées. Le but principal de cette thèse est d'améliorer SAT en distribué. Nos contributions s'articulent donc autour des deux approches de résolution parallèle du problèmes SAT (*portfolio* et « diviser pour mieux régner »)

dans un cadre massivement distribué.

Ce manuscrit est divisé en deux parties. Tout d'abord, nous introduisons les notions essentielles à la compréhension de ce document : le problème SAT et ses applications (chapitre 1). Ensuite, un état de l'art contenant les différents paradigmes de résolution séquentielle est établi, expliquant notamment les approches complètes du principe de résolution de Robinson au solveur SAT de nos jours (chapitre 2). Par la suite, une introduction au parallélisme montre les différentes possibilités qu'il nous offre (chapitre 3). Puis pour terminer cet état de l'art, nous présentons les différentes méthodes de résolution parallèle du problème SAT existant dans la littérature et détaillons les principales améliorations et évolutions apportées au fil des ans par les solveurs SAT parallèle (chapitre 4).

La seconde partie est consacrée à nos contributions. Celles-ci ont pour principal objectif la réalisation de solveurs SAT distribués performants. Ces derniers peuvent alors être employés avec une multitude de machines comme nous pouvons en trouver dans les fermes de calcul (*computers cluster*) ou en infonuagique (*cloud computing*).

Nos premières contributions exposées dans le chapitre 5 constituent les prémices de nos objectifs. Dans celles-ci, nous élaborons deux solveurs permettant de réaliser des divisions statiques (avant la recherche) et dynamiques (pendant la recherche) du problème initial en sous-problèmes. Le premier, nommé UCTSAT, est basé sur l'algorithme UCT (Finnsson 2012). Cette méthode est souvent utilisée pour les jeux (*General game playing*) et possède l'avantage de traiter le dilemme exploitation/exploration via un arbre de recherche. UCTSAT est alors une adaptation d'UCT à SAT. Plus précisément, c'est une décomposition statique qui profite de l'heuristique UCB (Auer et Ortner 2010) afin de pouvoir trier les sous-problèmes générés suivant leur prétendue utilité. La deuxième contribution présentée dans ce chapitre concerne SWARMSAT. C'est un solveur parallèle largement inspiré du solveur CUBEANDCONQUER (Heule *et al.* 2012) et est dédié au massivement parallèle. Cette première mouture peut utiliser comme *workers* l'un des deux solveurs de l'état de l'art GLUCOSE ou MINISAT.

Les travaux conduits sur ces deux solveurs nous ont amené à élaborer un nouveau schéma de décomposition dynamique nommé AMPHAROS (chapitre 6). Notre but étant la résolution du problème SAT dans un cadre massivement parallèle, il est important de proposer une architecture qui adapte sa stratégie en fonction du nombre de *workers* et de la nature du problème. À cette fin, nous proposons une approche qui utilise un algorithme adaptatif afin d'ajuster simultanément et dynamiquement le nombre de clauses partagées et le nombre de nouveaux cubes (sous-problèmes disponibles). Cela est possible grâce à l'utilisation d'une nouvelle mesure estimant si le processus de recherche doit être intensifié ou diversifié. Nous montrons que quand l'espace de recherche a besoin d'être diversifié (resp. intensifié), la mesure proposée détecte que le nombre de cubes (sous-problèmes) et le nombre des clauses partagées doivent être augmentés (resp. diminués). Dans AMPHAROS, un processus appelé MANAGER gère les cubes (sous-problèmes) et chaque solveur travaille sur la formule complète en induisant un de ces cubes. Par opposition aux autres solveurs « diviser pour mieux régner », notre approche peut faire travailler plusieurs solveurs sur le même sous-problème et ces derniers peuvent arrêter la recherche avant d'avoir trouvé une solution ou une contradiction. Cela permet d'éviter qu'un solveur ne passe trop de temps sur un même sous-problème trop difficile pour lui. Dans ce cas, le solveur demande au MANAGER un autre sous-problème qui peut provenir soit d'un autre cube existant, soit d'une subdivision d'un sous-problème trop difficile. Notre approche a pour originalité de laisser les solveurs sélectionner eux-mêmes les cubes qu'ils doivent résoudre. De plus, deux sortes de clauses sont échangées : les classiques clauses apprises et d'autres dépendant du cube. Nous avons nommé ces derniers littéraux (ou clauses) unitaires sous hypothèse. En effet, ce sont des littéraux qui sont propagés par quelques solveurs à un niveau de décision de l'hypothèse (du cube représentant le sous-problème). Le MANAGER récupère ces littéraux afin d'effectuer des techniques *inprocessing* et ainsi pouvoir renvoyer plus d'informations aux autres solveurs.

Bien que AMPHAROS ait obtenu de très bon résultats lors des expérimentations, nous avons détecté une anomalie au niveau de l'échange des clauses apprises. Plus précisément, nous mettons en évidence que l'encombrement du réseau par l'échange des clauses a un impact important sur leur efficacité. Pour remédier à ce problème, nous étudions dans le chapitre 7, l'impact des modèles de programmation parallèle sur cet échange. Ce chapitre a pour objectif d'apporter une solution novatrice aux possibles congestions du réseau dans un environnement massivement distribué.

Nous réalisons d'abord une étude expérimentale du solveur *multi-thread* SYRUP afin d'exposer son extensibilité. Puis, nous évaluons deux modèles de programmation distribués existants et déjà utilisés dans des solveurs SAT. Le premier est nommé le modèle de programmation pur par passage de messages et est utilisé par AMPHAROS et TOPOSAT (Ehlers *et al.* 2014). Le deuxième est appelé le modèle partiellement hybride et est utilisé par HORDESAT (Balyo *et al.* 2015). Nous montrons alors que ces schémas ont un impact direct sur les performances des solveurs. Grâce à ces résultats, nous introduisons un modèle complètement hybride à partir du solveur SYRUP qui n'a, à notre connaissance, jamais été appliqué à SAT. Nous comparons expérimentalement les différents modèles de programmation ainsi qu'une version distribuée de SYRUP nommée D-SYRUP basée sur le modèle de programmation complètement hybride contre deux solveurs de l'état de l'art. Ce modèle de programmation surpasse significativement les solveurs SAT de l'état de l'art TOPOSAT et HORDESAT sur un vaste ensemble d'instances. De plus, nous montrons expérimentalement qu'en pratique, ce schéma permet de partager plus de clauses sans pénaliser le solveur. En effet, dans notre nouveau modèle complètement hybride, le nombre de clauses partagées et utiles à la recherche est doublé par rapport à celui partiellement hybride sans que le nombre de propagations par seconde n'en soit affecté. Nos expérimentations semblent montrer que partager les clauses aussi vite que possible entraîne un impact positif sur l'efficacité des solveurs.

État de l'art

Chapitre 1

Le problème SAT

Sommaire

1.1 Logique propositionnelle **10**
1.1.1 Syntaxe 10
1.1.2 Sémantique 11
1.1.3 Formes normales 13
1.2 Théorie de la complexité **14**
1.3 Problème SAT **15**
1.3.1 Encodages 15
1.3.2 Fragments polynomiaux 16
1.4 Applications **17**
1.4.1 Exemples 18
1.5 Conclusion **21**

Tous les travaux ont un commencement composé des notions primordiales qui les accompagnent. Les nôtres consistent à résoudre des instances du problème SAT (pour problème de SATisfaisabilité booléenne). Ces instances sont exprimées grâce aux formules du langage de la logique propositionnelle et représentent une multitude d'applications scientifiques allant de la vérification de modèles de systèmes informatiques ou électroniques (Biere *et al.* 1999) aux problèmes mathématiques (Bennett et Zhang 2004a, Herwig *et al.* 2007, Hartley 1996) comme ceux de la théorie de Ramsey (Heule *et al.* 2016). Résoudre de telles instances du problème SAT permet alors de vérifier des propriétés dans de nombreux domaines.

Dans ce chapitre, nous présentons d'abord le cadre de la logique propositionnelle qui est à la base du problème SAT (section 1.1). Après avoir exposé les notions de la théorie de la complexité (section 1.2), qui mettent en avant l'importance de SAT puisqu'il est le premier problème à être prouvé $NP-Complet$ (Cook 1971), nous le définissons (section 1.3).

Dans la section 1.3, nous exposons les encodages qui permettent de traduire (Tseitin 1968) un problème quelconque (une formule de la logique propositionnelle) en une forme plus restreinte de cette logique appelée forme normale disjonctive (CNF). Cette forme définie aussi le problème SAT puisqu'elle permet de résoudre toutes les instances de SAT via des formules possédant la même structure (des conjonctions de clauses). Ensuite, nous présentons quelques familles d'instances possédant la particularité de pouvoir être résolues plus rapidement à cause de leurs structures particulières (problèmes 1-SAT, 2-SAT, Horn-SAT, ...).

Pour finir nous présentons le champ d'applications du problème SAT (section 1.4), notamment, nous décrivons précisément l'encodage de quatre problèmes applicatifs en SAT : les problèmes de la vérification de modèles bornés (Biere *et al.* 1999), de la coloration de graphes (Bar-Noy *et al.* 1998), du carré latin (Bennett et Zhang 2004b) et des triplets pythagoriciens booléens. Ce dernier problème à été récemment résolu grâce à SAT et apporte une preuve de sa résolution de 200 téra-octet de données. C'est, à ce jour, la preuve mathématique la plus longue et celle-ci n'est, bien sûr, pas lisible par un humain mais vérifiable par un ordinateur (Heule *et al.* 2016).

1.1 Logique propositionnelle

La logique propositionnelle (parfois appelée logique des propositions) est le fragment le plus simple de la logique mathématique classique. Il s'agit d'un langage formel pour exprimer des connaissances et raisonner avec celles-ci. C'est une vision de l'intelligence artificielle où tout raisonnement se modélise via un jeu de symboles. Dans cette logique, une proposition (ou variable) est une phrase déclarative munie d'une valeur de vérité qui est soit vraie, soit fausse, mais pas les deux : c'est le principe du tiers exclus logique.

1.1.1 Syntaxe

Dans un premier temps, nous définissons formellement la syntaxe de la logique propositionnelle. Pour cela, nous présentons le langage de la logique propositionnelle de la manière suivante.

Définition 1.1 (Morphologie de la logique propositionnelle)**.**
La **morphologie de la logique propositionnelle** est donnée par :

- un ensemble infini dénombrable de variables (aussi appelé symboles propositionnels) noté $\mathcal{V}$;
- un ensemble de connecteurs logiques usuels noté $\mathcal{C}$ où chaque élément c de $\mathcal{C}$ est muni d'une arité notée $arite(c)$ et est associé à une unique application ;
- des symboles $\perp$ et $\top$ représentant respectivement les constantes propositionnelles « faux » et « vrai » ;
- des symboles de ponctuation souvent notés « (», «) », « [» et «] ».

L'alphabet ne suffit pas à définir un langage. Il faut aussi définir les règles qui permettent de structurer ce dernier.

Définition 1.2 (Formule propositionnelle)**.** L'ensemble des **formules de la logique propositionnelle classique** $\mathcal{F}_{prop}$ est défini inductivement en appliquant un certain nombre de fois les règles suivantes :

- si $\alpha \in \{\mathcal{V} \cup \{\perp, \top\}\}$ alors $\alpha \in \mathcal{F}_{prop}$;
- si $\alpha \in \mathcal{F}_{prop}$ et $c \in \mathcal{C}$ avec $arite(c) = 1$ alors $c\ \alpha \in \mathcal{F}_{prop}$;
- si $\alpha_1 \in \mathcal{F}_{prop}$, $\alpha_2 \in \mathcal{F}_{prop}$ et $c \in \mathcal{C}$ avec $arite(c) = 2$ alors $\alpha_1\ c\ \alpha_1 \in \mathcal{F}_{prop}$;
- d'une manière générale, si $\alpha_1 \in \mathcal{F}_{prop}$, …, $\alpha_n \in \mathcal{F}_{prop}$ et $c \in \mathcal{C}$ avec $arite(c) = n$ alors $c\ (\alpha_1, \ldots, \alpha_n) \in \mathcal{F}_{prop}$.

Une formule appartenant à cet ensemble est alors dite formule bien formée.

Dans la suite de ce chapitre, la notation utilisée est préfixée mais nous pouvons en utiliser d'autres à condition qu'aucune ambiguïté n'apparaisse. Les valeurs booléennes sont représentées par l'ensemble $\mathbb{B} = \{faux, vrai\} = \{0, 1\} = \{\perp, \top\}$. Nous représentons une formule quelconque de la logique propositionnelle incluse dans $\mathcal{F}_{prop}$ par Σ. De plus, Les variables d'une formule bien formée Σ représentées par un ensemble infini dénombrable sont notées $\mathcal{V}_\Sigma$.

Exemple 1.1. Soient $\mathcal{V} = \{a, b, c, d\}$ un ensemble de variables propositionnelles et $\mathcal{C} = \{\vee, \wedge, \neg\}$ un ensemble de connecteurs logiques tel que $arite(\vee) = arite(\wedge) = 2$ et $arite(\neg) = 1$. La formule $\Sigma_1 = \{(a \vee b) \wedge (c\ \neg d)\}$ n'appartient pas à $\mathcal{F}_{prop}$, contrairement à $\Sigma_2 = \{(a \vee b) \wedge (c \vee d)\}$.

Dans la section suivante nous nous intéressons à la sémantique des formules bien formées, c'est-à-dire sous quelles conditions un énoncé est « vrai » ou « faux ».

1.1.2 Sémantique

La sémantique de la logique propositionnelle associe une signification à ses formules et explique les conditions qui rendent les formules vraies ou fausses. Cette sémantique est un morphisme : elle permet de relier les variables et les connecteurs de cette logique. De plus, les connecteurs sont vérifonctionnels : nous pouvons représenter leurs sens comme des fonctions prenant en entrée une ou des valeurs booléennes ($\{0, 1\}$) selon l'arité du connecteur, et donnent comme résultat une valeur booléenne.

Définition 1.3 (Interprétation)**.** Une **interprétation** $\mathcal{I}$ d'une formule propositionnelle Σ est une application de l'ensemble des variables de la formule $\mathcal{V}_\Sigma$ dans $\mathbb{B}$ qui attribue une valeur de vérité à chaque symbole propositionnel :

$$\mathcal{I} : \mathcal{V}_\Sigma \to \mathbb{B}$$

Le nombre d'interprétations possibles dépend du nombre de variables : avec $|\mathcal{V}_\Sigma|$ variables propositionnelles distinctes, il y a $2^{|\mathcal{V}_\Sigma|}$ interprétations possibles distinctes.

Exemple 1.2. Soit Σ une formule composée de deux variables $\mathcal{V}_\Sigma = \{a, b\}$. Dans ce cas, nous avons quatre interprétations différentes possibles : $\mathcal{I}_1 = \{a = 1, b = 1\}$, $\mathcal{I}_2 = \{a = 1, b = 0\}$, $\mathcal{I}_3 = \{a = 0, b = 1\}$ et $\mathcal{I}_4 = \{a = 0, b = 0\}$.

Définition 1.4 (Sémantique de la logique propositionnelle)**.** La **sémantique de la logique propositionnelle** $\mathcal{S}_\Sigma(\mathcal{I})$ d'une formule Σ par l'interprétation $\mathcal{I}$ est un élément de $\mathbb{B}$ défini inductivement par ces sous-formules α :

- si $\alpha = \bot$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \bot = 0$;
- si $\alpha = \top$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \top = 1$;
- si $\alpha \in \mathcal{V}$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \mathcal{I}(\alpha)$;
- si $\alpha = c(\alpha_1)$ avec $c \in \mathcal{C}$ et $arite(c) = 1$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \mathcal{F}_c(\mathcal{S}_{\alpha_1}(\mathcal{I}))$;
- si $\alpha = c(\alpha_1, \alpha_2)$ avec $c \in \mathcal{C}$ et $arite(c) = 2$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \mathcal{F}_c(\mathcal{S}_{\alpha_1}(\mathcal{I}), \mathcal{S}_{\alpha_2}(\mathcal{I}))$;
- d'une manière générales, si $\alpha = c(\alpha_1, \ldots, \alpha_n)$ avec $c \in \mathcal{C}$ et $arite(c) = n$ alors $\mathcal{S}_\alpha(\mathcal{I}) = \mathcal{F}_c(\mathcal{S}_{\alpha_1}(\mathcal{I}), \ldots, \mathcal{S}_{\alpha_n}(\mathcal{I}))$.

À partir d'une interprétation $\mathcal{I}$, nous pouvons déterminer la valeur de toutes formules en donnant un sens ($\mathcal{F}_c$) à chaque connecteur c de $\mathcal{C}$.

Définition 1.5 (Sémantique des connecteurs usuels de la logique propositionnelle)**.**

		Négation	Disjonction	Conjonction	Implication matérielle	Équivalence logique	Disjonction exclusive
x	y	$\alpha = \neg x$	$(x \vee y)$	$(x \wedge y)$	$(x \Rightarrow y)$	$(x \Leftrightarrow y)$	$(x \oplus y)$
		$\mathcal{F}_\neg(x)$	$\mathcal{F}_\vee(x, y)$	$\mathcal{F}_\wedge(x, y)$	$\mathcal{F}_\Rightarrow(x, y)$	$\mathcal{F}_\Leftrightarrow(x, y)$	$\mathcal{F}_\oplus(x, y)$
$vrai$	$vrai$	$faux$	$vrai$	$vrai$	$vrai$	$vrai$	$faux$
$vrai$	$faux$	$faux$	$vrai$	$faux$	$faux$	$faux$	$vrai$
$faux$	$vrai$	$vrai$	$vrai$	$faux$	$vrai$	$faux$	$vrai$
$faux$	$faux$	$vrai$	$faux$	$faux$	$vrai$	$vrai$	$faux$

TABLE 1.1 – Sémantique usuelle des opérateurs logiques.

Remarque 1.1. Dans ce manuscrit, la sémantique d'une interprétation de Σ (valeur booléenne obtenue via l'application de la sémantique sur une interprétation donnée) est notée $\mathcal{I}_\Sigma$ ou $\mathcal{I}(\Sigma)$.

Définitions 1.6 (Littéral, Littéral positif/négatif et Littéraux complémentaires)**.**

Un **littéral** est une variable propositionnelle ou bien sa négation. Autrement dit, un littéral associe ou pas le connecteur usuel de la négation. Soit ℓ un symbole propositionnel, alors ℓ est appelé **littéral positif**, $\neg\ell$ est appelé **littéral négatif**, ℓ et $\neg\ell$ sont des **littéraux complémentaires** et le complémentaire de ℓ est noté $\tilde{\ell}$.

Exemple 1.3. Soit a une variable, a et $\neg a$ sont des littéraux respectivement positif et négatif. Le complémentaire de $\neg a$ noté $\tilde{\neg a}$ est $\neg\neg a = a$.

Remarque 1.2. Une interprétation peut donc être naturellement représentée par un ensemble de littéraux. Par exemple, $\mathcal{I} = \{a, b, \neg c\} = \{a = 1, a = 1, c = 0\}$. De plus, nous pouvons étendre la notion d'interprétation à une formule Σ grâce à la sémantique : $\mathcal{I}_\Sigma = \mathcal{S}_\Sigma(\mathcal{I})$

Exemple 1.4. Soient $\Sigma = \{(a \vee b) \wedge (a \Rightarrow c)\}$ une formule propositionnelle et $\mathcal{I} = \{a, b, \neg c\}$ une interprétation de Σ. Nous avons $\mathcal{I}(\Sigma) = \mathcal{S}_\Sigma(\mathcal{I}) = (\top \vee \top) \wedge (\top \Rightarrow \bot) = \top \wedge \bot = \bot$.

Une interprétation n'est pas forcément définie sur l'ensemble des variables propositionnelles de la formule. Dans ce cas, l'interprétation peut être partielle ou incomplète, ce qui se définit formellement de la manière suivante :

Définitions 1.7 (Interprétation partielle/complète)**.** Soit Σ une formule propositionnelle, une interprétation $\mathcal{I}$ construite sur les variables de Σ notée $\mathcal{V}_\Sigma$ est dite :

- **partielle** si $|\mathcal{I}| < |\mathcal{V}_\Sigma|$;
- **complète** si $|\mathcal{I}| = |\mathcal{V}_\Sigma|$.

Il est aussi possible de prolonger une interprétation partielle vers une interprétation complète de la manière suivante :

Définition 1.8 (Prolongement d'une interprétation partielle)**.** Soient $\mathcal{I}_p$ et $\mathcal{I}_c$ respectivement une interprétation partielle et complète d'une formule propositionnelle Σ. Le prolongement de $\mathcal{I}_p$ par $\mathcal{I}_c$ est l'interprétation (partielle ou complète) $\mathcal{I}_{pro}$ tel que $\mathcal{I}_p \cup \mathcal{I}_{pro} = \mathcal{I}_c$ et qu'aucun complémentaires des littéraux de $\mathcal{I}_{pro}$ soient dans $\mathcal{I}_p$: $\forall \ell \in \mathcal{I}_{pro}, \tilde{\ell} \cap \mathcal{I}_p = \emptyset$.

Exemple 1.5. Soient les interprétations d'une formule de deux variables $\mathcal{I}_p = \{a\}$, $\mathcal{I}_{pro1} = \{a, b\}$ et $\mathcal{I}_{pro2} = \{\neg a, b\}$. $\mathcal{I}_{pro1}$ prolonge l'interprétation partielle $\mathcal{I}_p$ tandis que $\mathcal{I}_{pro2}$ ne la prolonge pas.

Définitions 1.9 (Modèle, Contre-modèle)**.** Une interprétation $\mathcal{I}$ est dite **modèle** d'une formule Σ, notée $\mathcal{I} \models \Sigma$, si la formule Σ est vraie dans $\mathcal{I}$, c'est à dire, $\mathcal{I}_\Sigma = \top$. En revanche, si la formule Σ est fausse dans $\mathcal{I}$, c'est-à-dire, $\mathcal{I}_\Sigma = \bot$, elle est dite **contre-modèle**.

Exemple 1.6. Soient la formule $\Sigma = \{(a \vee b) \wedge (a \vee c)\}$ ainsi que deux interprétations associées $\mathcal{I}_1 = \{a, b, c\}$ et $\mathcal{I}_2 = \{\neg a, b, \neg c\}$. Nous avons $\mathcal{I}_1(\Sigma) = (\top \vee \top) \wedge (\top \vee \top) = \top \wedge \top = \top$ et $\mathcal{I}_2(\Sigma) = (\bot \vee \top) \wedge (\bot \vee \bot) = \top \wedge \bot = \bot$. Par conséquent, l'interprétation $\mathcal{I}_1$ est un modèle de Σ alors que $\mathcal{I}_2$ est un contre-modèle.

Définitions 1.10 (Formule satisfaisable, insatisfaisable, valide)**.** Nous disons qu'une formule Σ est :

- **satisfaisable** (cohérente) si elle possède au moins un modèle : $\exists \mathcal{I}$ tel que $\mathcal{I}_\Sigma = \top$;

- **insatisfaisable** (incohérente, contradictoire) si elle possède aucun modèle : $\forall \mathcal{I}, \mathcal{I}_\Sigma = \bot$;
- **valide** (tautologie) si toutes ces interprétations sont des modèles : $\forall \mathcal{I}, \mathcal{I}_\Sigma = \top$.

Exemple 1.7. Les formules $\Sigma_1 = \{(a \vee b) \wedge (a \vee c)\}$, $\Sigma_2 = \{(a \vee \neg a) \wedge (a \vee c)\}$ et $\Sigma_3 = \{a \Rightarrow a\}$ sont respectivement satisfaisable, insatisfaisable et valide.

Remarque 1.3. Bien que les formules valides sont, dans notre situation, définies par le concept d'interprétation, leur vérité ne repose que sur leur structure, les variables ne sont pas analysées (interprétées).

Définitions 1.11 (Conséquence logique, Équivalence logique)**.**

Si tout modèle d'une formule Σ_1 est modèle d'une autre formule Σ_2 alors Σ_2 est une **conséquence logique** de Σ_1, noté $\Sigma_1 \models \Sigma_2$.

Les formules Σ_1 et Σ_2 sont **logiquement équivalentes**, noté $\Sigma_1 \equiv \Sigma_2$, si et seulement si $\Sigma_1 \models \Sigma_2$ et $\Sigma_2 \models \Sigma_1$.

1.1.3 Formes normales

Dans cette section, nous énonçons les différentes structures possibles d'une formule propositionnelle en fonction des connecteurs utilisés. Plus particulièrement, nous nous intéressons aux différentes formes normales d'une formule et nous exposons leurs relations.

Définitions 1.12 (Clause et Cube)**.**

Une **clause** est une disjonction finie de littéraux de la forme $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$.

Un **cube** (**terme** ou **monôme**) est une conjonction finie de littéraux de la forme $(\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n)$.

Définitions 1.13 (Clause unitaire, binaire, ternaire et n-aire)**.**

Une clause est **unitaire**, **binaire**, **ternaire** et **n-aire** si et seulement si elle est, respectivement, exactement composée de un, deux, trois et $n > 3$ littéraux différents.

Définitions 1.14 (Clause positive, négative et mixte)**.**

Une clause est **positive** (resp. **négative**) si et seulement si elle est constituée uniquement de littéraux positifs (resp. négatifs). Une clause composée de littéraux positifs et négatifs est une clause **mixte**.

Définitions 1.15 (Clause satisfaite, unisatisfaite et falsifiée)**.**

Une clause c est **satisfaite** par une interprétation $\mathcal{I}$ si au moins un de ces littéraux est affecté à vrai : $\exists \ell \in c$ tel que $\mathcal{I}(\ell) = vrai$. En conséquence, la clause **tautologique**, contenant deux littéraux complémentaires est **satisfaite**, elle est alors notée $\top$.

Une clause c est **unisatisfaite** par une interprétation $\mathcal{I}$ si un seul de ces littéraux est affecté à vrai et les autres à faux : $\exists \ell_i \in c$ tel que $\mathcal{I}(\ell_i) = vrai$ et $\forall \ell_j \in c$ avec $\ell_j \neq \ell_i$ tel que $\mathcal{I}(\ell_j) = faux$.

Une clause c est **falsifiée** par une interprétation $\mathcal{I}$ si tous ces littéraux sont affectés à faux : $\forall \ell \in c$ tel que $\mathcal{I}(\ell) = faux$. De plus, la clause **vide**, ne contenant aucun littéral, est **falsifiée**, elle est alors notée $\bot$.

Définition 1.16 (Clause subsumée)**.** Une clause $c_1 = (a_1 \vee a_2 \vee \cdots \vee a_n)$ **subsume** une clause $c_2 = (a_1 \vee a_2 \vee \cdots \vee a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$ si c_2 contient tous les littéraux de c_1.

Définitions 1.17 (Forme normale négative (NNF), conjonctive (DNF) et disjonctive (CNF))**.**

Nous distinguons trois formes normales particulières de formule de la logique propositionnelle :

- une formule est sous forme normale négative (NNF pour « *Negative Normal Form* ») si elle est exclusivement constituée de conjonctions, de disjonctions et de littéraux ;
- une formule est sous forme normale disjonctive (DNF pour « *Disjonctive Normal Form* ») si c'est une disjonction de termes ;
- une formule est sous forme normale conjonctive (CNF pour « *Conjonctive Normal Form* ») si c'est une conjonction de clauses.

Exemple 1.8. La formule $\Sigma_1 = \{(\neg a \vee b) \wedge (\neg c \vee d) \wedge (b \vee d \vee \neg c)\}$ est sous la forme normale conjonctive (CNF) tandis que la formule $\Sigma_2 = \{(a \wedge \neg b) \vee (c \wedge d) \vee (\neg b \vee d \wedge c)\}$ est sous la forme normale disjonctive (DNF).

Propriété 1.1. Toute formule bien formée de la logique propositionnelle peut être traduite sous une forme normale. De plus, la formule à traduire et celle obtenue sont équivalentes.

1.2 Théorie de la complexité

Une machine de Turing est un modèle mathématique de calcul définissant une machine abstraite (basée sur un modèle théorique) qui manipule un ensemble fini de symboles en changeant le contenu des cases d'un tableau infini selon des règles.

Définition 1.18 (Problème NP-complet)**.** Dans la théorie de la complexité, un **problème NP-complet** est un problème de décision appartenant à la fois à la classe NP et à la classe NP-difficile. Dans ce contexte, NP signifie « Non-déterministe Polynomial » car il peut être résolu par une machine de Turing non-déterministe en un temps polynomial. Un problème est NP-difficile si il est au moins aussi difficile que tous les autres problèmes de la classe NP.

Bien que toute solution donnée à un problème NP-complet peut être vérifié rapidement (en temps polynomial), il n'y a pas de moyen efficace connu pour trouver une solution. Autrement dit, le temps nécessaire pour résoudre le problème en utilisant des algorithmes connus augmente très rapidement (exponentiellement) en fonction de la taille du problème. Les problèmes appartenant au complémentaire de NP sont dit $coNP$. Le problème complémentaire de la satisfaisabilité (il existe une interprétation qui satisfait la formule) d'une formule est le problème vérifiant sa validité (toutes les interprétations satisfont la formule, ce qui est équivalent, il n'existe pas d'interprétation qui satisfait la négation de la formule).

La structure des formules propositionnelles a un impact colossal sur leur résolution. D'un coté, la satisfaisabilité d'une formule DNF peut être vérifiée en un temps linéaire : il suffit de vérifier si au moins une de ces conjonctions est satisfaisable. Une telle conjonction est satisfaisable si elle ne contient pas à la fois un littéral et son complémentaire. Plus encore, une formule DNF exprime directement tous ses modèles par sa structure. Cependant, vérifier la validité d'une formule DNF est difficile ($coNP$-complet).

À contrario, dans le cas d'une formule CNF, vérifier la satisfaisabilité est difficile (NP-complet) tandis que vérifier sa validité est facile (il suffit d'examiner chaque clause de la CNF). En d'autres mots, même si résoudre la satisfaisabilité d'une formule DNF est trivial, celle d'une formule CNF est amplement plus difficile (NP-complet). En effet, pour traduire une formule CNF en DNF, un nombre exponentiel de termes peut être construit au cours de la traduction, de sorte que la traduction elle-même ne peut pas s'exécuter en un temps polynomial. Clairement, une telle traduction consiste à résoudre la formule afin de trouver tous ses modèles.

Exemple 1.9. La formule $\Sigma_1 = \{(\neg a \land b) \lor (\neg a \land c)\}$ est satisfaisable et ses modèles sont $\{\neg a, b\}$ et $\{\neg a, c\}$. En revanche, la formule $\Sigma_1 = \{(\neg a \land a) \lor (\neg b \land b)\}$ est insatisfaisable.

1.3 Problème SAT

Grâce à ses nombreuses implications dans de nombreux domaines à la fois théorique et pratique, le problème SAT est devenu le couteau suisse des problèmes difficiles en informatique. Entrepris par les travaux de Cook (1971), celui-ci prouva son appartenance à la classe des problèmes NP-complet. Il fut le premier problème a être prouvé NP-complet, servant ainsi de référence à une énorme variété de problèmes complexes.

Définition 1.19 (Problème de décision)**.** Un **problème de décision** consiste à savoir si un problème énoncé est « vrai » ou « faux ».

Plus encore, de nombreux problèmes du monde réel ont été formalisés comme un problème de décision SAT à travers différentes techniques de codage. Comme, par exemple, les problèmes de vérification (matériels et logiciels), de planification ou de mathématique (voir section 1.4.1).

Définition 1.20 (Le problème SAT)**.** Le **problème SAT** (pour SATisfaisabilité d'une formule propositionnelle mise sous forme normale conjonctive) est un problème de décision qui consiste à déterminer si une formule CNF admet ou non un modèle.

Remarquons tout de même qu'il existe de nombreux travaux permettant de résoudre une formule n'étant pas de la forme CNF. Par exemple, nous pouvons citer les travaux de Christian *et al.* (2004) qui comparent le temps de résolution entre des formules propositionnelles quelconques (non CNF) et leurs variantes encodées en CNF. Les auteurs obtiennent de bons résultats via l'algorithme DPLL (cet algorithme est expliqué dans le chapitre 2, section 2.2.3, page 29). Ces techniques sont surtout efficaces sur certains problèmes structurés mais aujourd'hui, la forme CNF reste une norme très populaire. Les raisons de ce véritable engouement pour cette norme sont simples.

Premièrement, les heuristiques (techniques de résolution) pour la forme CNF sont valables sur tous les problèmes traduits en CNF, ce qui n'est pas le cas pour les heuristiques associées à une structure particulière et/ou un problème particulier. Deuxièmement, la normalisation des problèmes (sous un format appelé DIMACS représentant une CNF) a contribué à l'utilisation intensive des solveurs SAT dans la résolution d'une grande variété de problèmes. Pour finir, la traduction en temps polynomial d'une formule quelconque en CNF en a permis l'accessibilité de la norme CNF. De ce fait, résoudre la satisfaisabilité d'une formule quelconque revient à résoudre la formule CNF traduite. Nous allons maintenant expliquer cette traduction.

1.3.1 Encodages

Tout d'abord, une formule de la logique propositionnelle peut être transformée en CNF via certaines de ces règles (lois de De Morgan, lois distributives, . . .).

Exemple 1.10. Soit la formule $\Sigma = \{a \Rightarrow ((b \lor c) \land (\neg b \lor \neg c))\}$. Par la définition de l'implication, nous avons $\Sigma = \{\neg a \lor ((b \lor c) \land (\neg b \lor \neg c))\}$. Puis, en appliquant la loi distributive du connecteur « ou », nous obtenons $\Sigma = \{(\neg a \lor b \lor c) \land (\neg a \lor \neg b \lor \neg c)\}$.

Même si dans cet exemple la formule est compacte, cette méthode conduit généralement à une CNF de taille exponentielle. Pour pallier ce problème, la traduction en CNF est usuellement faite grâce à l'encodage de Tseitin (1968). Cette traduction ne donne pas une formule équivalente mais équisatisfaisable (satisfaisable si et seulement si la formule originale est satisfaisable). Elle possède l'avantage d'ajouter un nombre linaire de clauses grâce à l'ajout d'un nombre linéaire de nouvelles variables. La formule quelconque est d'abord traduite sous une forme NNF en un temps linéaire.

Exemple 1.11. Dans cet exemple, nous appliquons l'algorithme Tseitin (1968) sur la formule $\mathcal{F} = \{a \Rightarrow ((b \vee c) \wedge (\neg b \vee \neg c))\}$ de l'exemple 1.10. Tout d'abord, nous introduisons donc une nouvelle variable v_0 représentant la formule $\mathcal{F}$. Ensuite, nous créons une autre variable v_1 représentant la partie gauche $((b \vee c) \wedge (\neg b \vee \neg c))$ de l'implication de $\mathcal{F}$. Les définitions des variables grâce aux connecteurs de l'équivalence et de l'implication produisent les clauses de la nouvelle formule sous forme CNF :

$$v_0 \Leftrightarrow (a \Rightarrow v_1) = \begin{cases} \neg v_0 \vee \neg a \vee v_1 \\ v_0 \vee a \\ v_0 \vee \neg v_1 \end{cases}$$

De la même manière, nous avons :

$$v_1 \Leftrightarrow (v_2 \wedge v_3) = \begin{cases} \neg v_2 \vee \neg v_3 \vee v_1 \\ \neg v_1 \vee v_2 \\ \neg v_1 \vee v_3 \end{cases}$$

$$v_2 \Leftrightarrow (b \vee c) = \begin{cases} \neg v_2 \vee b \vee c \\ \neg v_2 \vee \neg c \\ \neg v_2 \vee \neg b \end{cases}$$

$$v_3 \Leftrightarrow (\neg b \vee \neg c) = \begin{cases} \neg v_3 \vee \neg b \vee \neg c \\ \neg v_3 \vee b \\ \neg v_3 \vee c \end{cases}$$

L'algorithme de Tseitin (1968) est très basique et manque d'optimisation. Notamment, plusieurs versions de cet encodage sont exposées dans la littérature, ajoutant plus ou moins de variables (Plaisted et Greenbaum 1986a, Markus *et al.* 2013). Hélas, ces traductions perdent certaines informations liées à la structure de la formule propositionnelle originale. Pour éviter ce problème, il existe d'autres encodages permettant de mieux considérer la structure d'origine comme l'encodage de Plaisted-Greenbaum (Plaisted et Greenbaum 1986b). De plus, l'objectif de certains travaux comme ceux de Jarvisalo *et al.* (2010) est de simplifier la formule CNF traduite avant la résolution : cela est appelé le pré-traitement (*preprocessing*) et est étudié dans le chapitre 2, section 2.3.7, page 53.

1.3.2 Fragments polynomiaux

Suivant certaines conditions, nous pouvons établir certaines contraintes à la définition générale du problème SAT. Typiquement, la structure des clauses d'une formule peut changer la complexité du problème. Ainsi, une multitudes de classes peuvent être résolues (dans le sens de la satisfaisabilité) en temps polynomial, voir linaire. Pour une formule CNF quelconque, il est donc intéressant de trouver un ou plusieurs fragments polynomiaux. Cette détection peut parfois être naturelle pour certaines classes mais pour d'autres, beaucoup plus laborieuse et coûteuse.

La taille des clauses

Définitions 1.21 (Le problème 2-SAT, 3-SAT, et k-SAT)**.** Le **problème k-SAT** (resp. **2-SAT** et **3-SAT**) est un problème de décision qui consiste à déterminer si une formule CNF composée uniquement de clauses

de taille k (resp. de taille 2 et de taille 3) admet ou non un modèle.

Les problèmes 1-SAT et 2-SAT sont polynomiaux tandis que les problèmes k-SAT avec $k > 2$ sont NP-complets (si les clauses ne représentent pas d'autres fragments polynomiaux). Par conséquence, remarquons que tous problèmes k-SAT avec $k > 3$ peuvent se réduire à 3-SAT.

Les clauses de Horn

Des autres classes polynomiales peuvent être définies grâce aux clauses de Horn. Cette classe est importante car elle fait le lien entre les clauses et la programmation logique. Nous pouvons citer le langage PROLOG qui, dans sa version initiale, n'acceptait que des clauses de Horn sous forme de régles telle que $a \wedge b \Rightarrow c$.

Définitions 1.22 (Clause de Horn et reverse-Horn)**.** Une clause dite **de Horn** (resp. **de reverse-Horn**) possède au plus un littéral positif (resp. négatif).

Définitions 1.23 (Le problème Horn-SAT, reverse-Horn-SAT et renameable-Horn)**.** Le **problème Horn-SAT** (resp. **reverse-Horn-SAT**) est un problème de décision qui consiste à déterminer si une formule CNF composée uniquement de clauses de Horn (resp. de reverse-Horn) admet ou non un modèle. Le **problème renameable-Horn-SAT** consiste à déterminer si une formule peut se traduire en une formule de Horn via un renommage des variables afin de déterminer sa satisfaisabilité.

Exemple 1.12. La formule $\Sigma = \{(a \vee \neg b \vee \neg c) \wedge (d \vee \neg a \vee \neg b)\}$ est une instance du problème Horn-SAT.

Trouver un modèle d'une formule Horn-SAT (resp. reverse-Horn-SAT) revient à appliquer la propagation unitaire (voir chapitre 2, section 2.2.3, page 29) sur tous les littéraux positifs (resp. négatifs) jusqu'à ce que tous ces littéraux soient éliminés. Ensuite, il suffit d'affecter tous les littéraux restants à faux (resp. à vrai). La propagation unitaire étant linaire, il va de soi que les problèmes de satisfaisabilité Horn-SAT et reverse-Horn-SAT le sont aussi (Minoux 1988, Dalal 1992, Rauzy 1995). Une formule peut être renameable-Horn en changeant la polarité de quelques variables afin d'obtenir une formule de Horn. Cette dernière classe est aussi détectée et résolu, en un temps linéaire (Lewis 1978).

Il existe de nombreux autres fragments polynomiaux. Parmi eux, nous pouvons citer les classes extended-Horn (Chandru et Hooker 1991), q-Horn (Boros *et al.* 1990) et linear-autarkies (Kullmann 1998).

1.4 Applications

Depuis son avènement, le problème SAT est très utilisé dans de nombreux domaines scientifiques et industriels. Dans ces champs d'applications, nous pouvons diviser les instances en deux catégories : celles aléatoires et celles structurées. La première est généralement liée au domaine scientifique et a pour but d'évaluer les algorithmes de résolution du problème SAT. Ces instances sont générées aléatoirement par des algorithmes dédiées et ne possèdent donc pas de réelle structure. En revanche, les instances issues de problèmes industriels sont majoritairement structurées. La plupart ont des formes et caractéristiques particulières, pouvant induire une forte difficulté de résolution. Parmi les plus importantes applications industrielles et scientifiques structurées, nous pouvons citer :

- la vérification de modèles bornés (BMC pour *Bounded Model Checking*) qui vérifie si le modèle d'un système (souvent informatique ou électronique) satisfait une propriété (Biere *et al.* 1999) ;

- la planification (Kautz et Selman 1996) qui peut être définie d'une manière générale, par la recherche d'une séquence d'actions à horizon borné atteignant un but prédéfini ;
- la vérification de logiciels (D'Silva *et al.* 2008), qui s'assure de l'exactitude d'un programme ;
- la résolution de problèmes combinatoires tel que les quasigroupes (Bennett et Zhang 2004a), les nombres de Ramsey et Van der Waerden (Herwig *et al.* 2007), les systèmes de Steiner (Hartley 1996), ...

Nous allons maintenant présenter l'encodage de quelques problèmes connus en problème SAT.

1.4.1 Exemples

Vérification de modèles bornés

Définition 1.24 (Problème de la vérification de modèles bornés)**.** Soit $\mathcal{P}$ une propriété, la **vérification de modèles** bornés consiste à savoir si il existe un état atteignable en k transitions qui ne satisfait pas la propriété dans un modèle (automate) donné.

La figure 1.1 expose une modélisation d'un compteur à deux bits. À titre d'exemple, nous étudions dans ce modèle si il existe un état atteignable en $k = 4$ transitions respectant la propriété $\mathcal{P}$ suivante : avons nous toujours un des deux bits à faux ? Pour cela, nous allons construire une formule propositionnelle $\Sigma(4)$ représentant ce problème.

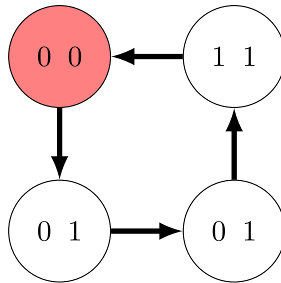


FIGURE 1.1 – Modélisation d'un compteur à deux bits. L'état initial, les états, et les transitions sont respectivement représentés par le nœud rouge, les noeuds et les flèches.

Nous définissons les variables propositionnelles l_i et r_i tel que $i < 4$. La variable l_i (resp. r_i) représente la valeur (0 ou 1) du bit à gauche (resp. à droite) dans l'état i. Ainsi, nous avons l'état initial $I = \neg l_0 \wedge \neg r_0$ (représenté par le nœud rouge). Les transissions sont représentées par l'ensemble des formules suivantes :

$$T = \begin{cases} (r_1 \Leftrightarrow \neg\, r_0) \wedge (r_2 \Leftrightarrow \neg\, r_1) \wedge (r_3 \Leftrightarrow \neg\, r_2) \\ (l_1 \Leftrightarrow \neg(l_0 \Leftrightarrow r_0)) \wedge (l_2 \Leftrightarrow \neg(l_1 \Leftrightarrow r_1)) \wedge (l_3 \Leftrightarrow \neg(l_2 \Leftrightarrow r_2)) \end{cases}$$

Ensuite, la propriété $\mathcal{P}$ (avons nous toujours un des deux bits à faux ?) est définie pour chaque état i tel que $P_i = \neg l_i \vee \neg r_i$. Pour finir, la formule propositionnelle est :

$$\Sigma(4) = \{I \wedge T \wedge (\neg P_0 \vee \neg P_1 \vee \neg P_2 \vee \neg P_3)\}$$

Après sa traduction en CNF puis sa résolution par un solveur SAT, $\Sigma(4)$ est satisfaisable car il existe un état qui ne satisfait pas la propriété (l'état où les deux bits sont tous les deux à vrais).

Coloration de graphe

Définition 1.25 (Problème de la coloration de graphes)**.** La **coloration de graphes** consiste à attribuer une couleur à chaque nœud d'un graphe tel que deux nœuds reliés par une arête soient d'une couleur différente.

Outre son intérêt théorique, le problème de la coloration de graphes a des applications pratiques dans différents domaines tels que la conception de circuits intégrés, la répartition des ressources distribuées, l'ordonnancement (Malafiejski *et al.* 2004), et plus particulièrement dans le domaine de la gestion des réseaux (Bar-Noy *et al.* 1998).

Considérons le graphe de la figure 1.2 composé de quatre nœuds et quatre arêtes. Pour représenter ce problème, nous créons des variables $x_{i,j}$. Une variable $x_{i,j}$ signifie que le nœud i est assigné à la couleur j. De plus, nous notons k, le nombre de couleurs disponibles. Nous avons trois contraintes à représenter :

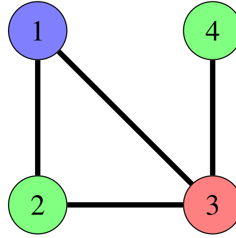


FIGURE 1.2 – Exemple d'une solution au problème de la coloration d'un graphe.

- Au moins une couleur à chaque noeud : $C_1 = \begin{cases} (x_{1,1} \vee x_{1,2} \vee \ldots \vee x_{1,k}) \wedge \\ (x_{2,1} \vee x_{2,2} \vee \ldots \vee x_{2,k}) \wedge \\ (x_{3,1} \vee x_{3,2} \vee \ldots \vee x_{3,k}) \wedge \\ (x_{4,1} \vee x_{4,2} \vee \ldots \vee x_{4,k}) \end{cases}$
- Au plus une couleur à chaque noeud : $C_2 = \begin{cases} (\neg x_{1,1} \vee \neg x_{1,2}) \wedge \ldots \wedge (\neg x_{1,k-1} \vee \neg x_{1,k}) \wedge \\ (\neg x_{2,1} \vee \neg x_{2,2}) \wedge \ldots \wedge (\neg x_{2,k-1} \vee \neg x_{2,k}) \wedge \\ (\neg x_{3,1} \vee \neg x_{3,2}) \wedge \ldots \wedge (\neg x_{3,k-1} \vee \neg x_{3,k}) \wedge \\ (\neg x_{4,1} \vee \neg x_{4,2}) \wedge \ldots \wedge (\neg x_{4,k-1} \vee \neg x_{4,k}) \end{cases}$
- Deux nœuds reliés ont des couleurs différentes : $C_3 = \begin{cases} (\neg x_{1,1} \vee \neg x_{2,1}) \wedge \ldots \wedge (\neg x_{1,k} \vee \neg x_{2,k}) \wedge \\ (\neg x_{2,1} \vee \neg x_{3,1}) \wedge \ldots \wedge (\neg x_{2,k} \vee \neg x_{3,k}) \wedge \\ (\neg x_{3,1} \vee \neg x_{1,1}) \wedge \ldots \wedge (\neg x_{3,k} \vee \neg x_{1,k}) \wedge \\ (\neg x_{3,1} \vee \neg x_{4,1}) \wedge \ldots \wedge (\neg x_{3,k} \vee \neg x_{4,k}) \end{cases}$

Comme le montre la figure 1.2, la conjonction $C_1 \wedge C_2 \wedge C_3$ est satisfaisable pour $k = 3$. En revanche, pour $k = 2$, elle est insatisfaisable.

Carré latin

Certains problèmes liés à la structure algébrique quasigroupe se modélisent en problèmes SAT. Le lecteur intéressé peut se référer aux travaux de Bennett et Zhang (2004b) qui présentent une étude des problèmes quasigroupes. Parmi eux, nous avons le problème du carré latin (parfois encore appelé quasigroupe). La structure sous-jacente des carrés latins présente une grande similitude par rapport à de nombreuses applications du monde réel telle que la planification et l'ordonnancement dans les systèmes d'exploitations et le multiplexage en longueur d'onde des réseaux de fibres optiques (Kumar *et al.* 1999). Remarquons que le sodoku est une version du carré latin plus contraignante.

Définition 1.26 (Problème du carré latin)**.** Le **carré latin** est une matrice de n lignes et n colonnes contenant n éléments. Ces éléments doivent apparaître une seule fois par ligne et par colonne.

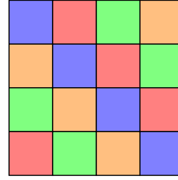


FIGURE 1.3 – Exemple d'une solution au problème d'un carré latin de taille 4.

Afin de représenter ce problème, nous créons des variables $x_{i,j,k}$. Une telle variable $x_{i,j,k}$ est vraie si la cellule i, j est assignée à la couleur k. De plus, nous notons n, le nombre de couleurs, de lignes et de colonnes. Comme dans l'exemple précédant, nous avons trois contraintes à représenter et la conjonction $C_1 \wedge C_2 \wedge C_3$ représente la formule propositionnelle :

- Des couleurs sont assignées aux cellules : $C_1 = \forall i, j(x_{i,j,1} \vee \ldots \vee x_{i,j,n})$
- Aucune couleur n'est répétée dans la même ligne : $C_2 = \forall i, k(\neg x_{i,1,k} \vee \vee x_{i,2,k}) \wedge (\neg x_{i,1,k} \vee \neg x_{i,3,k}) \wedge \ldots \wedge (\neg x_{i,1,k} \vee \neg x_{i,n,k}) \wedge \ldots \wedge (\neg x_{i,n,k} \vee \neg x_{i,n-1,k})$
- Aucune couleur n'est répétée dans la même colonne : $C_3 = \forall j, k(\neg x_{1,j,k} \vee \vee x_{2,j,k}) \wedge (\neg x_{1,j,k} \vee \neg x_{3,j,k}) \wedge \ldots \wedge (\neg x_{1,j,k} \vee \neg x_{n,j,k}) \wedge \ldots \wedge (\neg x_{n,j,k} \vee \neg x_{n-1,j,k})$

Le problème des triplets pythagoriciens booléens

Définition 1.27 (Problème des triplets pythagoriciens booléens)**.** Ce problème provient de la théorie de Ramsey. Il est définit comme le problème de décision demandant si il est possible de colorier chaque entier positif, noté x_i, soit en bleu, soit en rouge, de la sorte qu'aucun triplet de Pythagore $x_a^2 + x_b^2 = x_c^2$ tel que $a \neq b \neq c$ construit à partir des entiers x_i ne soit de la même couleur.

À titre d'exemple, considérons le problème des triplets pythagoriciens booléens jusqu'à dix entiers. Soient les variables x_i avec $i < 10$ telles que l'affectation de x_i est à vrai (resp. faux) si l'entier i est en bleu (resp. en rouge). Il n'y a qu'une seule sorte de contrainte à représenter : les triplets de Pythagore. Dans l'ensemble des entiers de 1 à 10, nous n'avons que deux triplets : $3^2 + 4^2 = 5^2$ et $6^2 + 8^2 = 10^2$. Nous devons donc avoir $x_3 \neq x_4 \neq x_5$ et $x_6 \neq x_8 \neq x_{10}$ afin de respecter l'énoncé. Notons bien que, x_3 peut être égal à x_4, c'est l'ensemble des trois variables qui ne doivent pas être égales. En logique propositionnelle, nous avons : $C_1 = \neg(x_3 \Leftrightarrow x_4 \wedge x_4 \Leftrightarrow x_5 \wedge x_3 \Leftrightarrow x_5) = \neg((\neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_5 \vee x_4) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_5 \vee x_3)) = (x_3 \wedge \neg x_4) \vee (x_4 \wedge \neg x_3) \vee (x_4 \wedge \neg x_5) \vee (x_5 \wedge \neg x_4) \vee (x_3 \wedge \neg x_5) \vee (x_5 \wedge \neg x_3) = (x_3 \vee x_4 \vee x_5) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_5)$. De la même manière, nous avons $C_2 = (x_6 \vee x_8 \vee x_{10}) \wedge (\neg x_6 \vee \neg x_8 \vee \neg x_{10})$. La figure 1.4 représente une solution possible (un des modèles) de la formule $C_1 \wedge C_2$.

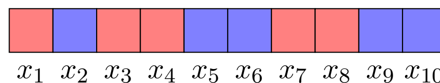


FIGURE 1.4 – Exemple d'une solution au problème des triplets pythagoriciens booléens dans dix entiers.

Les auteurs de Heule *et al.* (2016) prouvent que le problème est satisfaisable jusqu'à 7824 entiers mais qu'il est insatisfaisable avec 7825 entiers. Ils utilisent le parallélisme afin de résoudre ces deux

instances. Pour l'instance insatisfaisable de 7825 entiers, le temps réel de calcul a duré plus de deux jours. De plus, le solveur SAT a permis de donner une preuve et une cause de l'insastifaisabilité. Nous étudions la résolution parallèle de ce problème dans le chapitre 4, section 4.2.6, page 109.

1.5 Conclusion

Nous avons exposé dans ce chapitre, la logique propositionnelle, le problème SAT et ses applications. Nos travaux se basent principalement sur la résolution de ces applications. Elles sont représentées par des ensembles d'instances dans les compétitions. Plus précisément, nous essayons de résoudre via nos travaux les instances, de la compétions SAT de 2013 (Balint *et al.* 2013), de la SAT *race* 2015 (`https://baldur.iti.kit.edu/sat-race-2015/`) et celles de compétition SAT 2016 (`https://baldur.iti.kit.edu/sat-competition-2016/`). Dans le prochain chapitre, nous allons donc présenter les méthodes de résolution séquentielle permettant de trouver une solution (un modèle) ou de prouver l'insatisfaisabilité d'une formule mise sous forme CNF.

Chapitre

2

Résolution séquentielle du problème SAT

Sommaire

2.1 Approches complètes ou incomplètes 23
2.1.1 Approches incomplètes 24
2.1.2 Approches complètes 25
2.1.3 Comparaison 25
2.2 Genèse des solveurs SAT 26
2.2.1 Résolution 26
2.2.2 L'algorithme Davis - Putnam 28
2.2.3 L'algorithme de Davis - Logemann - Loveland 29
2.3 Les solveurs SAT 34
2.3.1 Apprentissage 34
2.3.2 L'algorithme CDCL 38
2.3.3 Choix de variable et choix de polarité 40
2.3.4 Structure de données paresseuses 45
2.3.5 Politiques de suppression des clauses apprises 48
2.3.6 Stratégies de redémarrage 51
2.3.7 Prétraitement 53
2.3.8 *Inprocessing* 57
2.4 Conclusion 57

Ce chapitre a pour but de présenter la résolution du problème SAT. Nous commençons par situer nos objectifs qui impliquent une résolution spécifique dite complète (section 2.1). Cela est dû au fait que nos instances cibles proviennent d'applications réelles (industrielles et mathématiques). Après avoir présenté les origines de la résolution du problème SAT (section 2.2), nous exposons le solveur SAT complet d'aujourd'hui (section 2.3). Pour finir, nous détaillons la totalité des composants intégrés dans un solveur SAT. Ces derniers apportent d'énormes progrès sur les problèmes de type industriel. En effet, les résultats des solveurs SAT sont très impressionnants (Marques-Silva et Sakallah 1996, Moskewicz *et al.* 2001a, Heule *et al.* 2016, Luo *et al.* 2017) et ne cessent de croître.

2.1 Approches complètes ou incomplètes

Les algorithmes complets permettent, en un temps fini, de déterminer une solution ou de prouver l'insatisfaisabilité d'une formule sous forme CNF. Ils s'appuient généralement sur le parcours en profondeur d'un arbre de recherche, où chaque nœud correspond à l'assignation d'une variable et chaque chemin correspondant à une interprétation des variables de la formule. L'objectif est alors de déterminer un chemin de la racine à une feuille - lequel représente une interprétation complète des variables de la formule - qui satisfait l'ensemble des contraintes du problème. Afin de ne pas explorer l'intégralité de cet arbre, la plupart des approches utilisent des méthodes de propagations de contraintes.

À l'inverse, les approches incomplètes, sont en générale incapables de répondre à l'insatisfaisabilité d'une formule. Ces approches effectuent le plus souvent un parcours de l'espace de recherche non systématique pendant un temps donné. Par conséquent, puisque ce type d'approche ne garantit pas un parcours complet de l'espace de recherche de la formule, et cela, quelle que soit la quantité de temps laissée à la méthode. Il est donc impossible d'affirmer qu'il n'y a pas de solution. Ainsi, une fois le temps imparti écoulé, deux cas peuvent se présenter : soit un modèle est trouvé et la procédure stoppe son exécution et retourne le modèle obtenu, soit la méthode retourne qu'elle n'a pas trouvé de solution et dans ce cas il est impossible de conclure.

2.1.1 Approches incomplètes

Il existe différents types d'approches incomplètes, les principales étant les techniques algorithmiques « génétiques » à base de population (Hao et Raphaël 1994), la « *survey propagation* » (Braunstein *et al.* 2005), la recherche à voisinage variable (Hansen *et al.* 2001), les algorithmes de colonies de fourmis (Dorigo et Stützle 2004), les algorithmes de parcours (Ow et Morton 1988) et la recherche locale. Dans cette section, nous présentons uniquement et globalement l'idée à l'origine de la recherche locale, celle-ci étant certainement la plus connue de toutes les méthodes incomplètes pour la résolution de SAT.

Algorithme 2.1 : Recherche Locale

Données : $\mathcal{F}$ une formule et $maxReparations$, le nombre maximum de réparations autorisées
Résultat : `vrai` si la formule $\mathcal{F}$ est satisfiable, `faux` s'il est impossible de conclure

1 **Début**
2 $\quad \mathcal{I} \longleftarrow$ une interprétation complète générée aléatoirement;
3 $\quad nbReparation = 0$;
4 $\quad$ **tant que** $(nbReparation < maxReparations)$ et $\mathcal{I}$ *n'est pas un modèle* **faire**
5 $\quad\quad$ **si** $\exists \mathcal{I}'$ *au voisinage de* $\mathcal{I}$ *telle que diff*$(\mathcal{F}, \mathcal{I}, \mathcal{I}') > 0$ **alors** $\mathcal{I} \longleftarrow \mathcal{I}'$;
6 $\quad\quad$ **sinon** $\mathcal{I} \longleftarrow$ interprétation choisie suivant un critère d'échappement ;
7 $\quad\quad$ $nbReparation \longleftarrow nbReparation + 1$;
8 $\quad$ **si** $\mathcal{I}$ *est un modèle* **alors retourner** `vrai` ;
9 $\quad$ **retourner** `faux` ;

Concrètement, une méthode de recherche locale (algorithme 2.1) s'appuie sur une fonction de voisinage. Dans le cadre de SAT, le voisinage d'une interprétation sont toutes les interprétations qui ne diffèrent que sur la valeur d'un seul littéral. Plus précisément, à partir d'une interprétation complète initiale, la méthode va se déplacer d'interprétation en interprétation voisine en permettant de réduire le nombre de clauses falsifiées. Dans le cas où il n'y a plus d'interprétation voisine permettant de diminuer le nombre de clauses falsifiées, et que l'interprétation courante n'est pas un modèle, alors un minimum local est atteint. Dans cette situation, il est nécessaire d'effectuer une perturbation dans l'interprétation courante. Cette perturbation est réalisée suivant une heuristique appelée critère d'échappement. Ainsi, c'est cette heuristique qui fait l'efficacité d'une méthode de recherche locale. Dans la littérature, de nombreux critères d'échappement ont été proposés (Mitchell *et al.* 1992, Selman *et al.* 1994). Cependant, en pratique les méthodes de recherche locale sont inefficaces sur les problèmes industriels. Dans ce cas les approches complètes restent la meilleure alternative.

2.1.2 Approches complètes

L'algorithme le plus simple et archaïque des approches complètes consiste à énumérer toutes les interprétations possibles et à calculer leurs valeurs de vérité via les connecteurs logiques jusqu'à obtenir un modèle. Si aucun modèle n'est trouvé, toutes les interprétations sont donc fausses et l'instance est donc insatisfaisable. Néanmoins, bien que cette méthode soit facile à implémenter, cette méthode est inefficace en pratique. En effet, pour une instance insatisfaisable composée de n variables, il est nécessaire de parcourir les 2^n interprétations possibles (Table 2.1).

a	b	$(a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee \neg b)$
$faux$	$faux$	$faux$
$faux$	$vrai$	$faux$
$vrai$	$faux$	$faux$
$vrai$	$vrai$	$faux$

TABLE 2.1 – Table de vérité de la formule $\Sigma = (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee \neg b)$.

Aujourd'hui, de nombreux algorithmes sont connus afin d'éviter le parcours de toutes ces interprétations. En particulier, le solveur complet d'aujourd'hui, apprend des informations provenant d'interprétations partielles falsifiées. Ces informations lui permettent de diminuer l'espace de recherche et ainsi d'éviter directement (sans devoir les parcourir) certaines interprétations connues pour être falsifiées.

2.1.3 Comparaison

Dans la pratique, les approches incomplètes sont plus performantes sur les instances aléatoires tandis que celles complètes sur les instances structurées. De ce fait, des méthodes hybrides sont apparues afin d'être efficaces à la fois sur des instances structurées et aléatoires. Emprunté à Audemard *et al.* (2009), le tableau 2.2 expose les résultats de trois solveurs différents (MINISAT (Eén et Sörenson 2004), SATHYS (Audemard *et al.* 2010; 2009) et WSAT (Selman *et al.* 1994)) représentant respectivement l'approche complète, hybride et incomplète. Le temps accordé à la résolution est de 1200 secondes et les instances proviennent de la compétition SAT de 2009. Ces résultats mettent en avant le fait que des approches complètes, hybrides et incomplètes sont généralement efficace sur une des trois principales catégories d'instances : conçues, industrielles et aléatoires. Les instances conçues et industrielles impliquent une structure dans leur forme tandis que celles aléatoires ne sont pas réellement structurées. Plus précisément, la plupart des instances industrielles ont des formes et caractéristiques particulières provenant du monde réel, pouvant induire une forte difficulté de résolution. Les instances conçues sont des instances construites sur mesure afin de représenter un problème concret. En ce qui concerne les instances aléatoires, elles sont générées de manière uniforme par des algorithmes dédiées.

Notre but étant la parallélisation massive d'un solveur SAT résolvant des instances provenant d'application ou industrielles, nous nous focalisons, dans la suite de ce manuscrit sur les approches complètes. La prochaine section est ainsi dédiée à l'évolution des différents algorithmes complets.

2.2 Genèse des solveurs SAT

Dans cette section, nous présentons différentes règles de résolutions et des algorithmes l'utilisant afin de résoudre des instances du problème SAT.

		Conçues		Industrielles		Aléatoires	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Complet	MINISAT	402	369	588	414	609	315
Hybride	SATHYS	340	174	473	266	930	17
Incomplet	WSAT	259	0	206	0	1012	0

TABLE 2.2 – Résultats des différentes approches (complètes, incomplètes et hybrides) sur différentes familles d'instances (Audemard *et al.* (2009))

2.2.1 Résolution

La résolutions est une règle d'inférence logique qui généralise le « modus ponens » en logique propositionnelle (Robinson 1965). Cette règle est principalement utilisée dans les systèmes de preuve automatiques.

Définition 2.1 (Résolution)**.** Soient $\alpha_1 = (a \vee b_1 \vee b_2 \vee \ldots \vee b_i)$ et $\alpha_2 = (\neg a \vee c_1 \vee c_2 \vee \ldots \vee c_j)$ deux clauses ayant en commun une variable a présente dans les deux clauses sous la forme de littéraux complémentaires. Une nouvelle clause $\alpha = (b_1 \vee b_2 \vee \ldots \vee b_i \vee \vee c_1 \vee c_2 \vee \ldots \vee c_j)$, appelée résolvante, peut être déduite par la suppression des occurrences du littéral a et $\neg a$ dans α_1 et α_2. Cette opération, notée $\eta[b, \alpha_1, \alpha_2]$, est appelée résolution et la clause produite est appelée résolvante.

Remarque 2.1. Dans la littérature et dans ce manuscrit, certaines techniques utilisant la résolution ou une de ces restrictions sont encore appelées résolution.

Exemple 2.1. Nous pouvons résoudre la clause $\alpha_1 = (a \vee b \vee \neg c)$ avec la clause $\alpha_2 = (\neg b \vee d)$ pour obtenir la résolvante $\eta[a, \alpha_1, \alpha_2] = (a \vee \neg c \vee d)$.

Un algorithmes complets les plus simples pour tester la satisfaisabilité d'une formule CNF est basée sur la règle de résolution (Robinson 1965). Il suffit d'appliquer la résolution sur une formule CNF jusqu'à obtenir une clause vide (insatisfaisable) ou jusqu'à ce qu'il ne soit plus possible d'en réaliser de nouvelles. La résolution est correcte mais incomplète, en d'autre terme, il n'est pas garanti de dériver toutes les clauses impliquées par une formule CNF. Toutefois, la résolution est réfutationnelement complète, c'est-à-dire, elle assure de retourner la clause vide si la formule est insatisfaisable (figure 2.1). Il est commun de représenter la séquence des résolutions effectuées par cet algorithme par un arbre inversé appelé alors arbre de dérivation. Dans le cas particulier où une clause vide est dérivée, cet arbre est dit de réfutation (figure 2.1).

Définition 2.2 (Affectation (conditionnement))**. Affecter** (ou **conditionner**, ou encore **simplifier**) dans une formule CNF Σ un littéral ℓ, noté $\Sigma_{|\ell}$, consiste à remplacer toutes les occurrences du littéral ℓ par la constante $vrai$ et toutes les occurrences du littéral $\neg\ell$ par la constante $faux$, puis simplifier. Cette démarche est équivalente à éliminer dans Σ toutes les clauses où le littéral ℓ apparaît et à supprimer chaque occurrence du littéral $\neg\ell$ dans les clauses le contenant.

Exemple 2.2. En affectant ℓ à $\Sigma = (\ell \vee a_1 \vee \ldots \vee a_i) \wedge \ldots \wedge (\neg\ell \vee b_1 \vee \ldots \vee b_j) \wedge \ldots \wedge (c_1 \vee \ldots \vee c_k)$, nous obtenons $\Sigma = (\top \vee a_1 \vee \ldots \vee a_i) \wedge \ldots \wedge (\bot \vee b_1 \vee \ldots \vee b_j) \wedge \ldots \wedge (c_1 \vee \ldots \vee c_k)$. Comme $\top \vee \alpha = \top$ et $\bot \vee \alpha = \bot$, nous avons $\Sigma = \top \wedge \ldots \wedge (b_1 \vee \ldots \vee b_j) \wedge \ldots \wedge (c_1 \vee \ldots \vee c_k)$. Pour finir, comme $\top \wedge \alpha = \alpha$, $\Sigma = (b_1 \vee \ldots \vee b_j) \wedge \ldots \wedge (c_1 \vee \ldots \vee c_k)$.

Exemple 2.3. Soit $\Sigma = \{(a \vee b \vee \neg c), (\neg a \vee c \vee \neg d), (\neg a \vee \neg b \vee d)\}$, En affectant a à faux, nous avons $\Sigma_{|\neg a} = (b \vee \neg c)$.

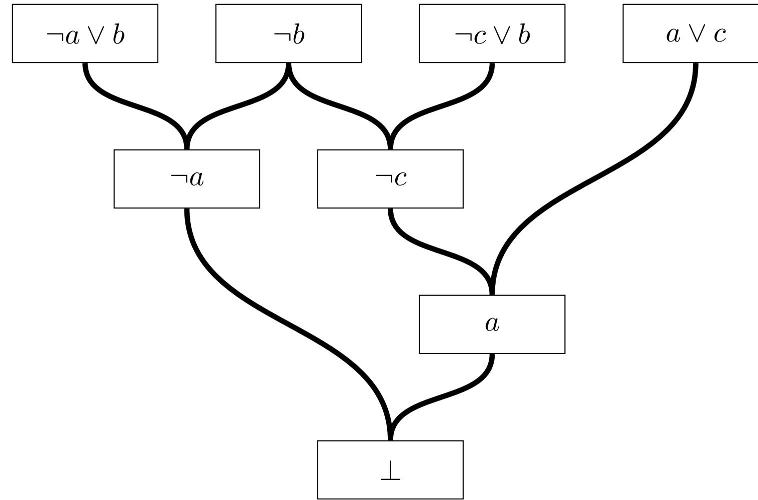


FIGURE 2.1 – Arbre de réfutation représentant des résolutions effectuées sur $(\neg a \vee b) \wedge (\neg c \vee b) \wedge \neg b \wedge (a \vee c)$.

Propriété 2.1 (Règle de division)**.**

Soit ℓ un littéral, une formule Σ est satisfaisable si et seulement si $\Sigma_{|\ell} \vee \Sigma_{|\neg\ell}$ est satisfaisable.

Remarque 2.2. Avec son propre algèbre et afin de représenter un circuit relié à un relais électromécanique, Shannon (1940) présenta cette règle de division jusqu'à arriver à une formule DNF. Cette règle est alors aussi appelée l'expansion de Shannon.

Cette propriété est très importante car elle est, avec la résolution, à la base des solveurs SAT modernes.

Définition 2.3 (Résolution appliquée à une variable ($\mathcal{V}$-résolution))**.** La **résolution appliquée à une variable** (**$\mathcal{V}$-résolution**) consiste à effectuer toutes les résolutions ayant en commun une variable v dans une formule CNF Σ. En d'autres mots, obtenir toutes les résolvantes de la forme $\eta[v, \alpha_i, \alpha_j]$ avec α_i et α_j des clauses quelconques de la formule Σ.

Exemple 2.4. Soit la formule $\Sigma = \{(\neg a \vee b) \wedge (\neg b \vee c)\} \wedge (\neg b \vee d)\}$, nous avons deux résolvantes sur la variable b tel que $\eta[b, (\neg a \vee b), (\neg b \vee c)] = \neg a \vee c$ et $\eta[b, (\neg a \vee b), (\neg b \vee d)] = \neg a \vee d$.

Définition 2.4 (Résolution de Davis et Putnam)**.**

La **résolution de Davis et Putnam** (Davis et Putnam 1960) consiste à effectuer la $\mathcal{V}$-résolution sur une variable v puis supprimer toutes les clauses contenant une occurrence de cette variable dans une formule CNF donnée.

Propriété 2.2 (Résolution de Davis et Putnam)**.** Les auteurs Davis et Putnam (1960) s'aperçoivent qu'effectuer la résolution de Davis et Putnam sur une variable (représentée par un littéral ℓ) est équivalent à faire la règle de division sur cette variable, c'est-à-dire, obtenir une formule équisatisfaisable de la forme $\Sigma_{|\ell} \vee \Sigma_{|\neg\ell}$. La résolution de Davis et Putnam peut donc être effectuée en calculant les clauses obtenues, d'une part, en affectant la variable à vrai ($\Sigma_{|\ell}$), et d'autre part, en affectant la variable à faux ($\Sigma_{|\neg\ell}$).

Exemple 2.5. À partir de l'exemple 2.4 ($\Sigma = \{(\neg a \vee b) \wedge (\neg b \vee c)\}$), la résolution de Davis et Putnam sur la variable b nous donne les clauses $\{(\neg a \vee b) \wedge (\neg b \vee c)\} \wedge (\neg a \vee c)\}$. La dernière clause est la résolvante de la résolution $\eta[b, (\neg a \vee b), (\neg b \vee c)]$. Ensuite, nous supprimons les clauses contenant la variable b, il ne nous reste donc plus qu'une seule clause $\{\neg a \vee c\}$.

À présent, calculons $\Sigma_{|b} \vee \Sigma_{|\neg b}$, nous avons $\Sigma_{|b} = c$ et $\Sigma_{|\neg b} = \neg a$. Ainsi, la règle de division $\Sigma_{|b} \vee \Sigma_{|\neg b} = \{\neg a \vee c\}$, produit la même clause que la résolution de Davis et Putnam.

Remarque 2.3. Une preuve de la relation entre la résolution de Davis et Putnam et de la règle de division est donnée dans l'article de Davis et Putnam (1960).

2.2.2 L'algorithme Davis - Putnam

Bien que l'article de Davis et Putnam (1960) possède tous les ingrédients de notre prochaine section (l'algorithme DPLL Davis *et al.* (1962)), il n'apporte aucune implémentation. Dans la littérature, l'algorithme de Davis-Putnam (DP) utilise uniquement la résolution. Il peut-être implémenté (Algorithme 2.2) grâce à un système de listes effectuant la résolution de Davis et Putnam sur un ordre donné de toutes les variables de la formule CNF (Dechter 1997).

Algorithme 2.2 : DP

Données : Σ une formule CNF et un ordre des variables $\prec_{\mathcal{V}}$
Résultat : SAT si la formule $\mathcal{F}$ est satisfaisable, UNSAT si elle est insatisfaisable

```
1  Début
2      pour chaque variable v ∈ V faire créer une liste vide L_v;
3      pour chaque clause c ∈ C faire
4          i = la première variable de la clause c suivant l'ordre ≺_V ;
5          L_i = L_i ∪ c ;
6      pour chaque variable v ∈ V suivant l'ordre ≺_V faire
7          si L_v n'est pas vide alors
8              pour chaque résolvante r = η[v, c_i, c_j] tel que c_i et c_j ∈ L_v faire
9                  si r est la clause vide ⊥ alors retourner UNSAT ;
10                 i = la première variable apparaissant dans la clause r suivant l'ordre ≺_V ;
11                 L_i = L_i ∪ r ;
12     retourner SAT ;
```

Exemple 2.6. La figure 2.2 applique l'algorithme 2.2 sur la formule $(\neg a \vee b) \wedge (\neg c \vee b) \wedge \neg b \wedge (a \vee c)$ avec l'ordre des variables $\prec_{\mathcal{V}} = \{a, b, c\}$. Dans cette figure, les clauses de la formule initiale sont grisées et chaque résolution est numérotée (1,2,3,4 et 5) indiquant ainsi quand elles ont été effectuées par l'algorithme DP. La clause vide prouvant l'insatisfaisabilité est engendrée par la boucle parcourant et créant les résolutions appliquées à la variable b ($\mathcal{V}$-résolution).

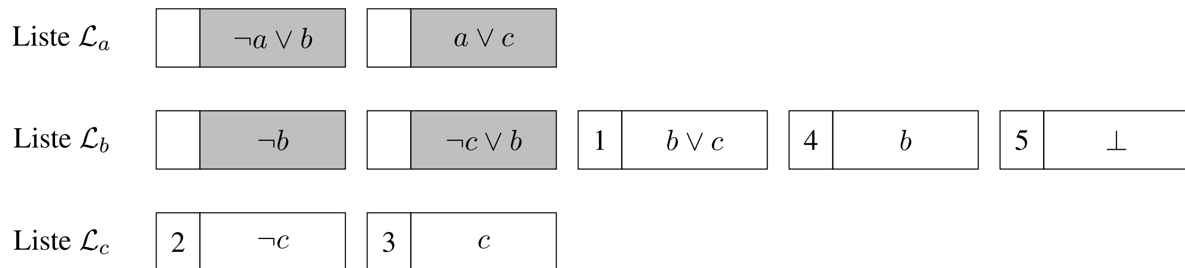


FIGURE 2.2 – Exemple de l'algorithme DP représentant les résolutions effectuées sur la formule $(\neg a \vee b) \wedge (\neg c \vee b) \wedge \neg b \wedge (a \vee c)$ avec l'ordre des variables $\prec_{\mathcal{V}} = \{a, b, c\}$.

En distribuant les résolutions sur un ordre donné des variables. L'algorithme DP permet de diminuer la complexité de la formule d'une variable à chaque étape. Les performances sont alors bien meilleures

que les algorithmes utilisant la résolution classique. Néanmoins, en pratique cette approche nécessite un espace exponentiel et elle est bien souvent inefficace.

2.2.3 L'algorithme de Davis - Logemann - Loveland

Dans l'article de Davis et Putnam (1960), les auteurs présentent plusieurs règles, certaines d'entre elles se révèlent primordiales à la résolution du problème SAT et sont utilisées dans l'algorithme DPLL. Parmi ces règles, nous pouvons citer :

- la résolution précédemment décrite (Définition 2.1) ;
- la règle des clauses unitaires (*Rule for the Elimination of One-Literal Clauses*) ;
- la règle des littéraux purs (*Affirmative-Negative Rule*) ;
- la résolution de Davis et Putnam (*Rule for Eliminating Atomic Formulas*), équivalente à la règle de division précédemment décrite (Propriété 2.1 et Définition 2.4).

Nous allons donc d'abord traiter les notions que nous n'avons pas encore présentées, à savoir, la règle des clauses unitaires et celle des littéraux purs. Par la suite, nous expliquons l'algorithme DPLL.

Résolution et propagation unitaire

Nous pouvons classer les différentes résolutions suivant leurs restrictions. Nous pouvons alors observer que la $\mathcal{V}$-résolution restreint la résolution à une variable. À présent, nous allons définir la résolution unitaire qui restreint la $\mathcal{V}$-résolution à une clause unitaire.

Définition 2.5 (Résolution unitaire)**.**

La **résolution unitaire**, pour une formule CNF donnée, consiste à effectuer la $\mathcal{V}$-résolution sur une variable v apparaissant dans une clause unitaire (contenant un seul littéral).

À la suite de cette résolution, il est courant de supprimer toutes les clauses où la variable v apparaît.

Exemple 2.7. Soit $\Sigma = \{(\neg a \vee \neg b) \wedge (a \vee c) \wedge (b \vee c) \wedge (\neg c \vee d) \wedge a\}$. En effectuant la résolution unitaire sur la clause unitaire a, nous ajoutons la clause $(\neg b)$ à la formule Σ. Puis, en supprimant toutes les clauses où la variable a apparaît, nous avons $\Sigma = \{\neg b \wedge (b \vee c) \wedge (\neg c \vee d)\}$.

Propriété 2.3 (Résolution unitaire)**.** Effectuer la résolution unitaire sur une variable v puis supprimer toutes les clauses où la variable v apparaît est équivalent à affecter son littéral associé ℓ, c'est-à-dire, calculer $\Sigma_{|\ell}$. Plus précisément, cela consiste à remplacer toutes les occurrences du littéral ℓ par la constante $vrai$ et toutes les occurrences du littéral $\neg\ell$ par la constante $faux$, puis simplifier. Cette démarche est équivalente à éliminer dans Σ toutes les clauses où le littéral ℓ apparaît et à supprimer chaque occurrence du littéral $\neg\ell$ dans les clauses le contenant.

Propriété 2.4 (Résolution unitaire)**.** Soit Σ une formule CNF contenant une clause unitaire composée du littéral ℓ. Les formules Σ et $\Sigma_{|\ell}$ sont équisatisfaisables, c'est-à-dire, Σ est satisfaisable si et seulement si $\Sigma_{|\ell}$ est satisfaisable.

Cette propriété découle du fait que les seules interprétations potentiellement capables de satisfaire la formule doivent satisfaire les littéraux appartenant aux clauses unitaires. La propagation unitaire est l'application répétée de cette simplification jusqu'à ce que la base de clauses ne contienne plus de clauses unitaires (c'est un point fixe). Formellement, nous avons :

Définition 2.6 (Propagation unitaire)**.** Soit Σ une formule, nous notons Σ^* la fermeture par propagation unitaire de Σ. Σ^* est définie récursivement comme suit :

- $\Sigma^* = \Sigma$ si $\nexists \alpha \in \Sigma$ telle que α est unitaire ;
- $\Sigma^* = \bot$ si Σ^* contient les deux clauses unitaires (ℓ) et $(\neg\ell)$;
- $\Sigma^* = \top$ si $\Sigma^* = \{\}$;
- $\Sigma^* = \Sigma^*_{|\ell}$ tel que ℓ est le littéral apparaissant dans une clause unitaire de Σ.

Propriété 2.5 (Propagation unitaire)**.** Soit Σ une formule CNF et Σ^* sa fermeture par propagation unitaire. Les formules Σ et Σ^* sont équisatisfaisables.

L'algorithme 2.3 exécute la propagation unitaire sur une formule donnée. Remarquons que celle-ci n'est pas complète, elle ne permet pas toujours de trouver la consistance d'une formule. Dans ce cas, nous disons que nous avons un point fixe : une formule close par la propagation unitaire.

Algorithme 2.3 : PROPAGATIONUNITAIRE (PU)

Données : Une formule CNF Σ^*
Résultat : SAT si la formule Σ^* est satisfaisable, UNSAT si elle est insatisfaisable ou Σ^* si un point fixe est obtenu

1 **Début**
2   **si** $\exists$ *une clause unitaire représentée par le littéral l dans* Σ^* **alors**
3     $\Sigma^* = \Sigma^*_{|\ell}$;
4   **si** $\Sigma^* = \{\emptyset\}$ **alors retourner** SAT ;
5   **si** Σ^* *contient les deux clauses unitaire* (ℓ) *et* $(\neg\ell)$ **alors retourner** UNSAT ;
6   **retourner** PROPAGATIONUNITAIRE (Σ^*) ;
7   **sinon**
    `// Un point fixe est obtenu`
8   **retourner** Σ^* ;

Exemple 2.8. Soit $\Sigma = \{a \wedge (\neg a \vee \neg b) \wedge (a \vee c) \wedge (b \vee c) \wedge (\neg c \vee d)\}$. Nous avons :

$\Sigma^* = \{\underline{a} \wedge (\underline{\neg a} \vee \neg b) \wedge (\cancel{a \vee c}) \wedge (b \vee c) \wedge (\neg c \vee d)\}$
$= \Sigma^*_{|a} = \{\underline{\neg b} \wedge (\underline{b} \vee c) \wedge (\neg c \vee d)\}$
$= \Sigma^*_{|a\ \neg b} = \{\underline{c} \wedge (\underline{\neg c} \vee d)\}$
$= \Sigma^*_{|a\ \neg b\ c} = \{\underline{d}\}$
$= \Sigma^*_{|a\ \neg b\ c\ d} = \{\}$, donc Σ est satisfaisable avec le modèle $\mathcal{I} = \{a, \neg b, c, d\}$.

Exemple 2.9. Soit $\Sigma = \{a \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b)\}$. Nous avons :

$\Sigma^* = \{\underline{a} \wedge (\underline{\neg a} \vee \neg b) \wedge (\underline{\neg a} \vee b)\}$
$= \Sigma^*_{|a} = \{(\neg b) \wedge (b)\} = \bot$. Σ est donc insatisfaisable.

Exemple 2.10. Soit $\Sigma = \{c \wedge (a \vee b \vee c \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg c \vee \neg e)\}$. Nous avons :

$\Sigma^* = \{\underline{c} \wedge (\cancel{a \vee b \vee c \vee d}) \wedge (a \vee \neg b \vee e) \wedge (\underline{\neg c} \vee \neg e)\}$
$= \Sigma^*_{|c} = \{\underline{\neg e} \wedge (a \vee \neg b \vee \underline{e})\}$
$= \Sigma^*_{|c\ \neg e} = \{(a \vee \neg b)\}$. Nous n'avons plus aucune clause unitaire, c'est un point fixe.

Littéraux purs

La seconde règle présentée permet de simplifier la formule lorsqu'un littéral apparaît dans la formule CNF mais pas son complémentaire. Dans ce cas ce littéral est dit pur, ce qui s'exprime formellement de la manière suivante :

Définition 2.7 (Littéral pur)**.**

Un littéral ℓ est dit **littéral pur** (ou littéral monotone) pour un ensemble de clauses Σ si il apparaît uniquement positivement ou uniquement négativement dans Σ, c'est-à-dire, si son littéral complémentaire $\neg\ell$ n'apparaît pas dans Σ.

Propriété 2.6 (Règle du littéral pur)**.** Soient Σ une formule CNF et ℓ_p un littéral pur de Σ. La formule Σ est équivalente à la formule $\Sigma_{|\ell_p}$. Supprimer toutes les clauses où le littéral pur ℓ_p apparaît nous donne donc une formule équisatisfaisable.

Exemple 2.11. Soit la formule $\Sigma = \{(a \vee b) \wedge (\neg b \vee \underline{\neg c}) \wedge (\neg a \vee b \vee \underline{\neg c})\}$. Le littéral $\neg c$ est pur, donc $\Sigma_{|\neg c} = \{(a \vee b)\}$.

Algorithme

Dans la littérature, les algorithmes DP et DPLL ont souvent été confondus car la totalité des ingrédients de DPLL était déjà dans l'article de DP (Davis et Putnam 1960). Le principal inconvénient de l'approche DP est lié à la complexité spatiale de celle-ci. Les clauses résolvantes générées par la $\mathcal{V}$-résolution sont trop nombreuses pour être gardées en mémoire. Pour pallier ce problème, en 1962, Martin Davis, George Logemann et Donald Loveland (Davis *et al.* 1962) proposent une procédure appelée DPLL. L'idée principale est d'appliquer la règle de division plutôt que de faire la $\mathcal{V}$-résolution. Rappelons que cela est équivalent (Propriété 2.2).

La méthode DPLL, décrit par l'algorithme 2.4, est en fait un algorithme de recherche arborescent de type « *depth-first search* » (recherche en profondeur d'abord) avec retours arrières. Étant donnée une formule Σ, cette procédure consiste à choisir un littéral ℓ de Σ (ligne 8) et à décomposer la formule Σ en deux sous-formules $\Sigma_{|\ell}$ et $\Sigma_{|\neg\ell}$. Ce principe, appelé séparation, est basé sur la règle de division : une formule Σ est consistante si et seulement si les formules $\Sigma_{|\ell}$ ou $\Sigma_{|\neg\ell}$ sont consistantes. Une fois cette décomposition matérialisée, la procédure DPLL consiste à tester la validité de la première sous-formule, si elle est consistante alors le problème est consistant, sinon la seconde formule est testée (ligne 9). Afin d'éviter l'exploration inutile de branches de l'arbre de recherche, les simplifications précédemment décrites sont opérées (la propagation unitaire et la règle des littéraux purs).

La figure 2.3 représente le déroulement de l'algorithme DPLL sur la formule Σ grâce à un arbre binaire où les branches (flèches) à gauche (resp. à droite) représentent une affectation à vrai (resp. à faux) de la variable choisie. L'ordre des affectations représentées par les nœuds et les flèches de l'arbre est celui de la récurrence (Algorithme 2.4, ligne 9) et est numéroté dans chaque nœud (sauf la racine) de la première affectation à la dernière.

Un point crucial dans l'efficacité de la procédure DPLL est le choix de la variable pour la règle de division. Ce choix, fait via une *heuristique de choix de variable* a fait l'objet de nombreuses études et la variable ainsi choisie est communément appelée variable (ou littéral) de décision. Afin d'affiner plusieurs définitions à venir (conflit, *backtrack*, ...), nous allons introduire quelques notions et notations permettant de représenter une partie de la recherche effectuée par les algorithmes du problème SAT.

Algorithme 2.4 : DPLL

Données : Σ une formule CNF
Résultat : `vrai` si la formule est consistante, sinon `faux`

1 **Début**
2 **si** $\Sigma = \{\emptyset\}$ **alors retourner** *vrai* ;
3 **si** $\Sigma = \bot$ **alors retourner** *faux* ;
4 **si** $\exists$ *une clause unitaire représentée par le littéral l dans* Σ **alors**
5 **retourner** DPLL $(\Sigma_{|\ell})$;
6 **si** $\exists$ *un littéral pur* ℓ *dans* Σ **alors**
7 **retourner** DPLL $(\Sigma_{|\ell})$;
8 $\ell \longleftarrow$ `HeuristiqueDeChoixDeVariable(`Σ`)` ;
9 **retourner** (DPLL $(\Sigma_{|\ell})$ *ou* DPLL $(\Sigma_{|\neg\ell})$);

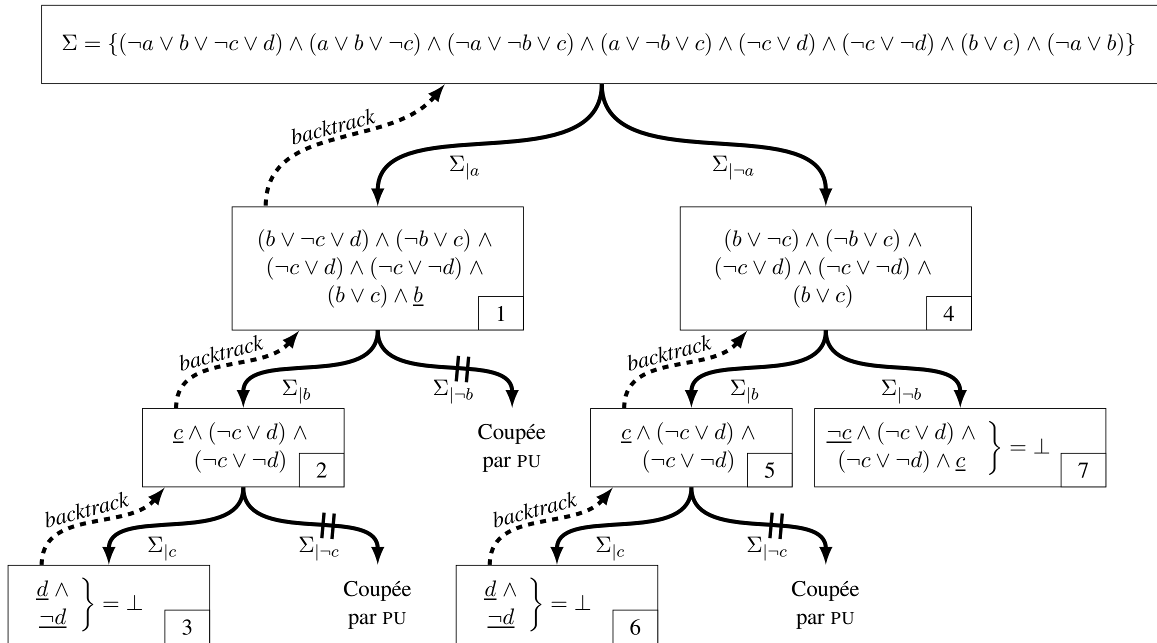


FIGURE 2.3 – Déroulement de l'algorithme DPLL via un arbre binaire de recherche sur Σ prouvée insatisfaisable.

Définition 2.8 (Décision et niveau de décision)**.**

Dans un algorithme de résolution du problème SAT, une **décision** est un synonyme à l'affectation d'une variable ou d'un littéral via l'heuristique de choix de variable (ceux de la propagation unitaire ne compte pas). Nous décidons le littéral ℓ signifie $\Sigma_{|\ell}$.

Le **niveau de décision d'un algorithme** est le nombre de décision courant effectué via l'heuristique de choix de variable. Notons que le niveau de décision zéro est le moment où aucune décision n'a encore été prise.

Le **niveau de décision d'une variable**, d'un littéral ou encore d'une décision est le niveau de décision de l'algorithme au moment où elle est affectée (par la PU ou l'heuristique de choix de variable). Si celle ci n'est pas encore affectée, son niveau est indéfini et est noté $\emptyset$.

Dans ce manuscrit, nous notons respectivement, $\mathcal{D}_{level}$, $\mathcal{D}_{level}(v)$ et $\mathcal{D}_{level}(\ell)$ le niveau courant d'un

algorithme, d'une variable et d'un littéral.

Exemple 2.12. Dans la figure 2.3, toutes les variables ont été décidées plusieurs fois à un moment donné. À titre d'exemple, au moment ou l'algorithme est dans le nœud numéro six ($d \wedge \neg d = \bot$), le niveau de décision de l'algorithme est $\mathcal{D}_{level} = 2$ et ceux des variables sont $\mathcal{D}_{level}(a) = 1$, $\mathcal{D}_{level}(b) = 2$, $\mathcal{D}_{level}(c) = 2$ et $\mathcal{D}_{level}(d) = \emptyset$ car cette dernière n'a pas été affectée.

Définitions 2.9 (Séquence de décisions/propagations)**.**

Une **séquence de décisions** de Σ est noté $\mathcal{S}_d = \{d_1, d_2, \ldots, d_n\}$ avec d_n les littéraux décidés.

Une **séquence de propagations** notée $\mathcal{S}_p = \langle d, \{p_1, p_2, \ldots, p_j\}\rangle$ est obtenue à partir de Σ tel que :

- d est soit une décision, soit le symbole $\emptyset$ lorsqu'aucune décision n'a encore été prise ;
- p_j sont les littéraux propagés (clauses unitaires) par la propagation unitaire :
  - sur $\Sigma_{|d}$, si d est une décision ;
  - sur Σ, sinon (quand $d = \emptyset$).

Une **séquence de décisions/propagations** est une interprétation $\mathcal{I}$ représentant plusieurs séquences de propagations et une séquence de décisions notée :

$\mathcal{I} = \langle \emptyset, \{x_{01}, x_{02}, \ldots, x_{0i}\}\rangle\langle d_1, \{x_{11}, x_{12}, \ldots, x_{1j}\}\rangle \ldots \langle d_n, \{x_{n1}, x_{n2}, \ldots, x_{nk}\}\rangle$ avec n le niveau de décision de d_n tel que :

- $\{x_{01}, x_{02}, \ldots, x_{0i}\}$ sont les littéraux propagés par la PU sur Σ ;
- $\{x_{11}, x_{12}, \ldots, x_{1j}\}$ sont le littéraux propagés par la PU sur $\Sigma_{|d_1}$;
- $\{x_{n1}, x_{n2}, \ldots, x_{nk}\}$ sont le littéraux propagés par la PU sur $\Sigma_{|d_1 \ldots d_n}$;

Définition 2.10 (Conflict)**.** Lorsqu'une interprétation est falsifiée par un algorithme, cela est appelé un **conflit** car deux littéraux ℓ et $\neg\ell$ entrent en conflit.

Définition 2.11 (*Backtrack*)**.** Quand un conflit survient, un ***backtrack*** (flèche avec des tirés sur la figure 2.3) est appliqué afin de retourner vers une branche de l'arbre binaire de recherche encore non explorée. Dans notre algorithme 2.4, cela est fait naturellement via la récurrence. Néanmoins, dans les algorithmes itératifs, une structure de données (pile de propagations) doit être mis en place afin de réaliser ces *backtracks*. Celle ci garde l'historique des décisions et des propagations, quand un *backtrack* survient, il suffit juste d'effacer la dernière séquence de propagations.

Exemple 2.13. L'algorithme DPLL de la figure 2.3 effectue plusieurs séquences de décisions/propagations :

- $\mathcal{I}_1 = \langle \emptyset, \{\emptyset\}\rangle\langle a, \{b, c, d, \neg d\}\rangle$ (Nœuds 1, 2 et 3) et *backtrack* ;
- $\mathcal{I}_2 = \underline{\langle \emptyset, \{\emptyset\}\rangle}\langle \neg a, \{\emptyset\}\rangle\langle b, \{c, d, \neg d\}\rangle$ (Nœuds 4, 5 et 6) et *backtrack* ;
- $\mathcal{I}_3 = \underline{\langle \emptyset, \{\emptyset\}\rangle\langle \neg a, \{\emptyset\}\rangle}\langle \neg b, \{c, \neg c\}\rangle$ (Nœuds 4 et 7) et UNSAT.

Les séquences de décisions/propagations $\mathcal{I}_1$, $\mathcal{I}_2$ et $\mathcal{I}_3$ de l'exemple précédant représentent chacune un conflit. Les deux conflits engendrés par $\mathcal{I}_1$ et $\mathcal{I}_2$ impliquent des *backtracks* tandis que quand $\mathcal{I}_3$ est affecter, la formule est prouvée insatisfaisable. Les parties de $\mathcal{I}_1$, $\mathcal{I}_2$ et $\mathcal{I}_3$ soulignées représentent le travail déjà accompli et à ne pas refaire grâce au *backtracks*.

Dans cette figure, nous observons clairement l'intérêt des méthodes de simplifications. Plus précisément, l'utilisation de la propagation unitaire permet d'éviter le parcours d'un nombre important d'interprétations inutiles (flèches barrées sur la figure). Chaque feuille de l'arbre est soit coupée par

la propagation unitaire (PU sur la figure), soit prouvée inconsistante (par deux clauses unitaires opposées : $\ell \wedge \neg\ell = \bot$). Ces résultats sont remontés jusqu'à la racine par la récurrence afin de prouver Σ insatisfaisable.

Au delà de ces simplifications, l'avantage majeur de DPLL est d'utiliser directement la propriété de la division sans produire toutes les résolvantes de la résolution. Contrairement à DP, cette manière de diviser permet à DPLL d'avoir une complexité en polynomiale en espace. Toutefois, son temps de réponse reste exponentiel (Crawford et Auton 1993). Nous allons maintenant étudier le solveurs SAT complet d'aujourd'hui.

2.3 Les solveurs SAT

Ce que nous appelons aujourd'hui, solveur SAT, c'est l'algorithme CDCL accompagné d'améliorations notables renforçant considérablement son efficacité. Celles-ci ont été apportées pas à pas dans la littérature et sont encore aujourd'hui très prisées et étudiées. Introduit par Marques-Silva et Sakallah (1996) et amélioré par Moskewicz *et al.* (2001a), l'algorithme CDCL ajoute la notion d'apprentissage à DPLL, qui permet d'apprendre des conflits (des erreurs du passé) afin de ne pas obtenir certains conflits semblables (éviter les prochaines erreurs). Avant de présenter globalement l'algorithme CDCL et ses composants internes, nous allons d'abord détailler l'apprentissage.

2.3.1 Apprentissage

L'objectif de l'apprentissage est de trouver une clause permettant d'éviter de reproduire le même échec ailleurs dans l'arbre de recherche. Pour déduire une de ces clauses dites apprises, nous devons trouver et analyser l'ensemble des littéraux responsables d'un conflit (Marques-Silva et Sakallah 1996, Bayardo Jr. et Schrag 1997).

Exemple 2.14. Soit la formule Σ suivante composée de six clauses et neuf variables :

$$\begin{array}{lll} c_1 = (x_1 \vee x_2) & c_2 = (x_1 \vee x_3 \vee x_7) & c_3 = (\neg x_2 \vee \neg x_3 \vee x_4) \\ c_4 = (\neg x_4 \vee x_5 \vee x_8) & c_5 = (\neg x_4 \vee x_6 \vee x_9) & c_6 = (\neg x_5 \vee \neg x_6) \end{array}$$

Avec la séquence de décisions $\mathcal{S}_d = \{\neg x_7, \neg x_8, \neg x_9, \neg x_1\}$, nous avons :

$\Sigma = \{(x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee \cancel{x_7}) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5 \vee x_8) \wedge (\neg x_4 \vee x_6 \vee x_9) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7} = \{(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5 \vee \cancel{x_8}) \wedge (\neg x_4 \vee x_6 \vee x_9) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8} = \{(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_4 \vee x_6 \vee \cancel{x_9}) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8\ \neg x_9} = \{(\cancel{x_1} \vee x_2) \wedge (\cancel{x_1} \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_4 \vee x_6) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8\ \neg x_9\ \neg x_1} = \{\cancel{x_2} \wedge \cancel{x_3} \wedge (\cancel{\neg x_2} \vee \cancel{\neg x_3} \vee x_4) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_4 \vee x_6) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8\ \neg x_9\ \neg x_1\ x_2\ x_3} = \{\cancel{x_4} \wedge (\cancel{\neg x_4} \vee x_5) \wedge (\cancel{\neg x_4} \vee x_6) \wedge (\neg x_5 \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8\ \neg x_9\ \neg x_1\ x_2\ x_3\ x_4} = \{\cancel{x_5} \wedge x_6 \wedge (\cancel{\neg x_5} \vee \neg x_6)\}$

$\Sigma_{|\neg x_7\ \neg x_8\ \neg x_9\ \neg x_1\ x_2\ x_3\ x_4\ x_5} = \{x_6 \wedge \neg x_6\} = \bot$

Ce conflit est obtenu à partir d'une séquence de décisions/propagations, représentée formellement par :

$\mathcal{I} = \langle \emptyset, \{\emptyset\}\rangle \langle \neg x_7, \{\emptyset\}\rangle \langle \neg x_8, \{\emptyset\}\rangle \langle \neg x_9, \{\emptyset\}\rangle \langle \neg x_1, \{x_2, x_3, x_4, x_5, x_6, \neg x_6\}\rangle$

Imaginons que nous voulons refaire cette séquence de décisions sans obtenir ce conflit. L'idée la plus simple est d'ajouter la clause $(x_7 \vee x_8 \vee x_9 \vee x_1)$ à la formule initiale. En effet, une fois les trois premières décisions effectuées, x_1 sera propagé, permettant ainsi d'éviter ce conflit. Notons tout de même, que l'algorithme DPLL aurait fait un simple *backtrack* vers x_1.

Bien que la méthode apportée par l'exemple précédant peut générer des clauses apprises, celles-ci sont très longues et chacune ne représente qu'un seul conflit. Remarquons que des travaux utilisent quand même cette manière de générer des clauses apprises quand celles-ci sont assez courtes (Knuth 1998). Une manière d'analyser plus en profondeur les conflits est basée sur l'analyse d'un graphe d'implications (Marques-Silva et Sakallah 1996). Ce graphe dirigé acyclique (DAG) permet de représenter les dépendances entre les clauses et les assignations obtenues par propagation des littéraux unitaires.

Définitions 2.12 (Clause raison et explication(s) d'un littéral)**.**

Soient Σ une formule, $\mathcal{I}$ une interprétation partielle obtenue par propagation à partir de Σ en appliquant la séquence de décisions $\mathcal{S}_d$ et ℓ un littéral de Σ.

La **clause raison** de ℓ, notée, $ori(\ell)$ est la clause à l'origine de la propagation. Le littéral associé à une décision ne possède pas de clause raison car il n'est pas propagé par la propagation unitaire. Formellement, nous avons :

- si $\ell \in \mathcal{S}_d$ alors $ori(\ell) = \bot$;
- sinon $ori(\ell)$ est une clause de Σ tel que :
  - $\ell \in ori(\ell)$ et $\mathcal{I}(\ell) = \top$;
  - et $\forall \ell' \in ori(\ell)$, tel que $\ell' \neq \ell$, nous avons $\mathcal{I}(\ell') = \bot$ avant que $\mathcal{I}(\ell) = \top$ suivant l'ordre des affectations dans $\mathcal{I}$.

L'ensemble des **explications** de ℓ, noté $exp(\ell)$, sont les littéraux $\neg\ell'$ tel que $\forall \ell' \in ori(\ell)$ avec $\ell' \neq \ell$. Autrement dit, si $ori(\ell) = \{\ell'_1 \vee \ldots \vee \ell'_n \vee l\}$ tel que $\ell \neq \ell'_n$, alors $exp(\ell)$ est l'ensemble $\{\neg\ell'_1, \ldots, \neg\ell'_n\}$. De plus, si $ori(\ell) = \bot$ alors $exp(\ell) = \{\emptyset\}$.

Remarque 2.4. Il est possible qu'un littéral soit propagé par plusieurs clauses raisons. Toutefois, il est usuel de n'en considérer qu'une seule.

Exemple 2.15. Dans l'interprétation de l'exemple 2.14, nous avons $ori(x_6) = c_5$ avec $exp(x_6) = \{x_4, \neg x_9\}$ et $ori(x_3) = c_2$ avec $exp(x_3) = \{\neg x_7, \neg x_1\}$.

Définition 2.13 (Graphe d'implications)**.** Soient Σ une formule, $\mathcal{I}$ une interprétation partielle, $\mathcal{N}$ un ensemble de nœuds et $\mathcal{A}$ un ensemble d'arêtes. Le graphe d'implications associé à Σ et $\mathcal{I}$ est un DAG $G = (\mathcal{N}, \mathcal{A})$ tel que :

- $\mathcal{N} = \{x \in \mathcal{I}\}$, c'est-à-dire un nœud pour chaque littéral de $\mathcal{I}$, de décision ou propagé ;
- $\mathcal{A} = \{(x, y)$ tel que $x \in \mathcal{I}, y \in \mathcal{I}$ et $x \in exp(y)\}$.

La figure 2.4 est le graphe d'implications de l'exemple 2.14.

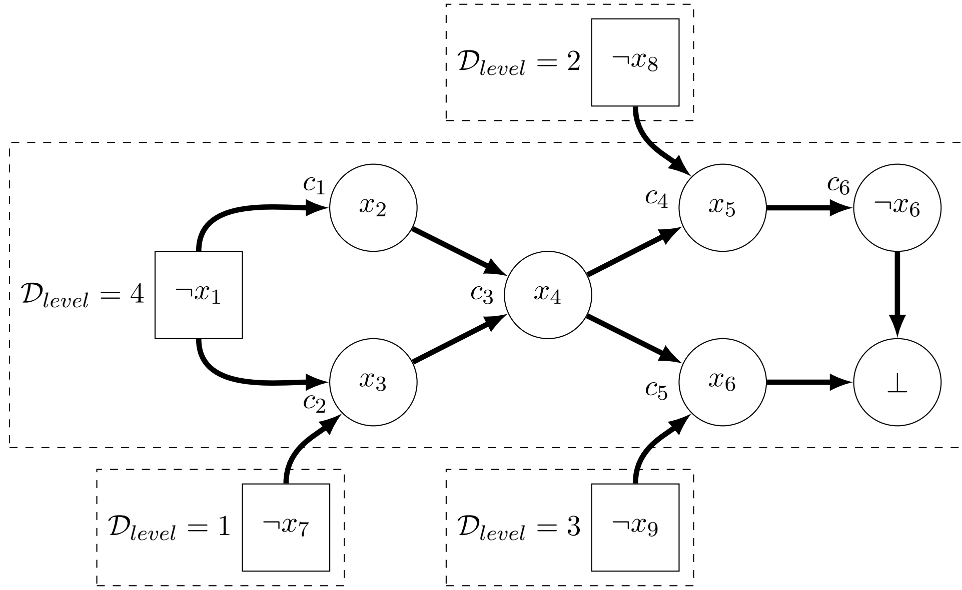


FIGURE 2.4 – Graphe d'implications obtenu à partir de la formule $\Sigma = \{(x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_7) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_5 \vee x_8) \wedge (\neg x_4 \vee x_6 \vee x_9) \wedge (\neg x_5 \vee \neg x_6)\}$ et l'interprétation $\mathcal{I} = \langle \emptyset, \{\emptyset\}\rangle\langle \neg x_7, \{\emptyset\}\rangle\langle \neg x_8, \{\emptyset\}\rangle\langle \neg x_9, \{\emptyset\}\rangle\langle \neg x_1, \{x_2, x_3, x_4, x_5, x_6, \neg x_6\}\rangle$ de l'exemple 2.14. Les décisions et les propagations sont respectivement représentées par les nœuds du graphe formant des carrés et des cercles. Chaque nœud est annoté par la clause raison ($ori()$) responsable de la propagation associée. Le conflit se trouve sur la variable x_6 et est représenté par le nœud $\bot$.

Définitions 2.14 (Ordre dans $G = (\mathcal{N}, \mathcal{A})$, nœud précédent et suivant)**.**

Soit un graphe d'implications $G = (\mathcal{N}, \mathcal{A})$ généré à partir d'une formule Σ et d'une interprétation partielle $\mathcal{I}$. L'ensemble des nœuds $\mathcal{N}$ est muni d'une **relation d'ordre totale** définie par l'ordre des affectations de $\mathcal{I}$. Un nœud $x \in \mathcal{N}$ est dit **précédent** (resp. **suivant**) d'un nœud $y \in \mathcal{N}$ si et seulement si il existe un chemin de x à y (resp. de y vers x). L'ensemble des nœuds précédents (resp. suivants) un nœud x est noté $prec(x)$ (resp. $suiv(y)$).

Définition 2.15 (Nœud dominant)**.** Un nœud $x \in \mathcal{N}$ **domine** un nœud $y \in \mathcal{N}$ si et seulement si :

- $\mathcal{D}_{level}(x) = \mathcal{D}_{level}(y)$ et ;
- $\forall z \in \mathcal{N}$ tel que $\mathcal{D}_{level}(z) = \mathcal{D}_{level}(x)$ et $z \in prec(x)$, tous les chemins de z vers y passent par x.

Exemple 2.16. Dans la figure 2.4, le nœud x_4 domine x_6 tandis que x_2 ne le domine pas.

Définitions 2.16 (point d'implication unique (UIP), premier et dernier UIP)**.** Soient Σ une formule, $\mathcal{I}$ une interprétation conflictuelle et $G = (\mathcal{N}, \mathcal{A})$ le graphe d'implications généré à partir de Σ en fonction de $\mathcal{I}$. Le nœud x est un **point d'implication unique** si et seulement si x domine le conflit. Les UIP peuvent être ordonnés en fonction de leur distance avec le conflit. Le **premier UIP** (F-UIP pour « *First Unique Implication Point* ») est l'UIP le plus proche du conflit tandis que le **dernier UIP** (L-UIP pour *« Last Unique Implication Point »*) est le plus éloigné, c'est-à-dire, le littéral de décision affecté au niveau du conflit.

Exemple 2.17. La figure 2.4 contient deux UIP, le nœud x_4 est le premier UIP tandis que le nœud $\neg x_1$ est le dernier UIP.

Définition 2.17 (Clause assertive)**.** Soient Σ une formule, $\mathcal{I}$ une interprétation obtenue par propagation unitaire et $\mathcal{D}_{level}$ le niveau de décision courant. Une clause α de la forme $(a_1 \vee \ldots \vee a_n \vee b)$ est dite

assertive si et seulement si $\mathcal{I}(\alpha) = \perp$, $\mathcal{D}_{level}(x) = \mathcal{D}_{level}$ et $\forall a_n$, $\mathcal{D}_{level}(a_n) < \mathcal{D}_{level}(x)$. Le littéral x est appelé littéral *assertif.*

Définition 2.18 (Preuve par résolution basée sur les conflits)**.** Soient Σ une formule, $\mathcal{I}$ une interprétation conflictuelle obtenue par propagation unitaire et $\mathcal{D}_{level}$ le niveau de décision courant. Une preuve par résolution basée sur les conflits est une séquence de clauses $\langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ obtenue via des $\mathcal{V}$-résolutions inductivement tel que :

- $\sigma_1 = \eta[x, ori(x), ori(\neg x)]$ tel que x et $\neg x$ sont conflictuels ;
- $\forall 1 \leq i \leq k$, $\sigma_i = \eta[y, \sigma_{i-1}, ori(\neg y)]$ telle que $y \in \sigma_{i-1}$, $\mathcal{D}_{level}(y) = \mathcal{D}_{level}$ et y est le noeud le plus proche du conflit suivant l'ordre de $\mathcal{N}$ dans $G = (\mathcal{N}, \mathcal{A})$.
- σ_k est une clause assertive.

Remarque 2.5. Le calcul de $G = (\mathcal{N}, \mathcal{A})$ n'est pas nécessaire dans la preuve par résolution basée sur les conflits car l'ordre de $\mathcal{N}$ est simplement l'ordre des affectations dans $\mathcal{I}$.

La preuve par résolution nous permet de calculer une ou plusieurs clauses dite assertives à partir d'un conflit. Ces clauses correspondent aux UIP du graphe d'implications associé. Elles sont générées en effectuant des résolutions sur les variables du niveau courant entre les clauses responsables du conflit (utilisées durant la propagation unitaire) ou/et leurs résolvantes suivant l'ordre des affectations jusqu'à obtenir une clause contenant un seul littéral du dernier niveau de décision. L'application de la définition 2.18 nous permet d'avoir la clause assertive associée au premier UIP. Après le calcul de cette dernière, il suffit d'appliquer encore la deuxième étape de la définition 2.18 (refaire des résolutions) jusqu'à l'obtention d'une nouvelle clause assertive représentant un autre UIP. Quand le littéral assertif de cette dernière est la décision du niveau courant, c'est le dernier UIP. Remarquons que le premier UIP peut être le dernier quand il n'y en a qu'un seul.

La plupart des solveurs SAT ne garde que la clause assertive représentant le premier UIP, celle-ci est alors appelée clause apprises.

Exemple 2.18. Considérons de nouveau la formule Σ et l'interprétation partielle $\mathcal{I}$ de l'exemple 2.14. La preuve par résolution basée sur les conflits est la suivante :

Nous effectuons une résolution sur x_6 car c'est le conflit :
$\sigma_1 = \eta[x_6, c_5, c_6] = \neg x_4^4 \vee \cancel{x_6^4} \vee x_9^3 \vee \neg x_5^4 \vee \cancel{\neg x_6^4} = \neg x_4^4 \vee \neg x_5^4 \vee x_9^3$
Puis sur x_5 car ce dernier est affectée après x_4 :
$\sigma_2 = \eta[x_5, \sigma_1, c_4] = \neg x_4^4 \vee \cancel{\neg x_5^4} \vee x_9^3 \vee \cancel{\neg x_4^4} \vee \cancel{x_5^4} \vee x_8^2 = \underline{\neg x_4^4 \vee x_9^3 \vee x_8^2}$ F-UIP
Nous continuons sur x_4 pour avoir une autre clause assertive :
$\sigma_3 = \eta[x_4, \sigma_2, c_3] = \cancel{\neg x_4^4} \vee x_9^3 \vee x_8^2 \vee \neg x_2^4 \vee \neg x_3^4 \vee \cancel{x_4^4} = x_9^3 \vee x_8^2 \vee \neg x_2^4 \vee \neg x_3^4$
Puis sur x_2 :
$\sigma_4 = \eta[x_2, \sigma_3, c_1] = x_9^3 \vee x_8^2 \vee \cancel{\neg x_2^4} \vee \neg x_3^4 \vee x_1^4 \vee \cancel{x_2^4} = x_9^3 \vee x_8^2 \vee \neg x_3^4 \vee x_1^4$
Et sur x_3 car ce dernier est affectée après x_1 :
$\sigma_5 = \eta[x_3, \sigma_4, c_2] = x_9^3 \vee x_8^2 \vee \cancel{\neg x_3^4} \vee x_1^4 \vee \cancel{x_1^4} \vee \cancel{x_3^4} \vee x_7^1 = \underline{x_1^4 \vee x_9^3 \vee x_8^2 \vee x_7^1}$ L-UIP

Définition 2.19 (*Backjump*)**.** Soient Σ une formule, $\mathcal{I}$ une interprétation partielle conflictuelle obtenue par propagation unitaire et α une clause assertive de la forme $(a_1 \vee \ldots \vee a_n \vee b)$ déduite à partir de l'analyse du conflit. Le niveau de ***backjump*** $\mathcal{D}_{backjump} = max\{\mathcal{D}_{level}(\neg a_1), \ldots, \mathcal{D}_{level}(\neg a_n)\}$ est un niveau de retour arrière correct à effectuer. De plus, à ce niveau, la clause assertive est unitaire et propage directement le littéral assertif b. Un *backjump* peut effectuer un saut plus grand dans l'arbre de recherche (de plusieurs niveaux), contrairement à un *backtrack* qui effectue un saut d'un seul niveau de décision. La clause assertive associée permet donc d'éviter d'explorer une partie de l'arbre de recherche.

Exemple 2.19. Le niveau de *backjump* des clauses assertives $\neg x_4^4 \vee x_9^3 \vee x_8^2$ (F-UIP) et $x_1^4 \vee x_9^3 \vee x_8^2 \vee x_7^1$ (L-UIP) est $\mathcal{D}_{backjump} = 3$. Après le retour arrière effectué, le F-UIP propage directement le littéral assertif $\neg x_4$ tandis que le L-UIP propage directement x_1.

Cette simple propriété montre pourquoi le premier UIP habituellement considéré dans les schémas d'apprentissage classique est plus puissant que les autres UIP.

Propriété 2.7. Soit α une clause assertive obtenue par l'analyse de conflit représentant le F-UIP. Le niveau de *backjump* de α est optimal. En d'autre mot, toutes autres clauses assertives β déduite de la même analyse de conflit, représentant les autres UIP, sont telles que $\mathcal{D}_{backjump}(\beta) \geq \mathcal{D}_{backjump}(\alpha)$ (Audemard *et al.* 2008b).

Exemple 2.20. Toujours sur l'exemple 2.14, le premier UIP représentant le conflit « protège » plus d'interprétations partielles de ce conflit que le dernier UIP.

Ainsi, en pratique, la plupart des solveurs SAT modernes apprennent uniquement la clause assertive associée au premier UIP. De nombreux travaux sont basés sur le processus d'analyse de conflits. Par exemple, Sörensson et Biere (2009) proposent une approche qui consiste à continuer d'effectuer des résolutions si celles-ci n'augmentent pas la taille de la clause assertive. D'autres essaient de découvrir des clauses sous-sommées durant l'analyse de conflits (Han et Somenzi 2009, Hamadi *et al.* 2009b). Une autre approche, proposée par Audemard *et al.* (2008a), tente d'étendre la notion de graphe d'implications afin de considérer certaines clauses satisfaites par la formule. Une dernière approche, proposée par Nadel et Ryvchin (2010), cherche à améliorer la hauteur du saut en effectuant un retour arrière plus grand que celui proposé par la clause assertive et à affecter et propager l'ensemble des littéraux de celle-ci. Notons que d'autres travaux dérivent des clauses apprises dites bi-assertives en traversant séparément le graphe d'application à partir d'un littéral x et son complémentaire $\neg x$ représentant un conflit (Jabbour *et al.* 2013).

Pour terminer, à chaque fois qu'une clause assertive est apprise, celle ci est ajoutée dans une base. Ainsi, les solveurs SAT modernes propagent en plus des clauses de la formule initiale, des clauses apprises. De plus, chacune des clauses propageant un littéral représente une étape de résolution. De ce fait, nous pouvons obtenir une preuve de l'insastifaisabilité. Néanmoins, à cause d'un nombre de clauses apprises trop élevé, deux problèmes majeurs s'ajoutent à leur utilisation : l'utilisation de la mémoire et le temps utilisé pour la propagation unitaire. Nous allons à présent décrire globalement l'algorithme CDCL et examiner comment ces deux problèmes sont traités.

2.3.2 L'algorithme CDCL

L'algorithme CDCL ne pouvant pas être représenté facilement par un algorithme récursif, une version itérative de celui-ci est donnée (Algorithme 2.5). Dans celui-ci, deux concepts de DPLL ne sont pas repris par CDCL, notamment, grâce à l'apprentissage :

- Plutôt que de faire un *backtrack* quand il y a un conflit, une analyse de conflit a lieu, une clause est apprise et un *backjump* est effectué (ligne 9 à 11 et ligne 17 à 20). Ces clauses apprises sont donc aussi, en plus des clauses de la formule initiale, utilisées par la propagation unitaire (ligne 8) ;
- La règle de division d'une formule est toujours appliquée mais l'algorithme CDCL choisi uniquement un seul côté. En d'autres mots, il n'explore plus les deux branches ($\Sigma_{|l} \vee \Sigma_{|\neg l}$) systématiquement et une heuristique de choix de polarité (section 2.3.3) est donc ajoutée à l'heuristique de choix de variable (section 2.3.3 et ligne 22 à 25 de l'algorithme 2.5).

Algorithme 2.5 : CDCL

```
   Données : Σ une formule CNF
   Résultat : vrai si la formule est satisfaisable, sinon faux
 1 Début
 2   L_pu ⟵ {∅};                                  /* Liste de littéraux à propager */
 3   learnt ⟵ {∅};                                /* Ensemble de clauses apprises */
 4   I_p ⟵ {∅};                                      /* Interprétation partielle */
 5   D_level ⟵ 0 ;                                        /* Niveau de décision */
 6   Σ ⟵prétraitement(Σ) ;                      /* Prétraitement de la formule */
 7   tant que vrai faire
 8     α ⟵ PropagationUnitaire(Σ ∪ learnt, I_p, L_pu);
 9     si α est un conflit alors
10       β ⟵analyseConflit(Σ ∪ Δ, I_p, α); /* β est une clause apprise */
11       si β = ⊥ alors retourner UNSAT ;
12       learnt ⟵ learnt ∪ β;
13       si redémarrage() alors
14         I_p ⟵ {∅} ; D_level ⟵ 0 ;
15         backjump(Σ ∪ Δ, I_p, D_level)
16       sinon
17         ℓ_a ⟵ littéralAssertif(β);
18         L_pu ⟵ L_pu ∪ ℓ_a ;
19         D_level ⟵calculBackjump(I_p, D_level, β);       /* En fonction de β */
20         backjump(Σ ∪ Δ, I_p, D_level);                  /* mise à jour de I_p */
21     sinon
         // La PU n'a pas obtenue de conflit, c'est un point fixe
22       x ⟵ heuristiqueDeChoixDeVariable(Σ ∪ Δ, I_p);
23       si toutes les variables sont affectées alors retourner SAT ;
24       ℓ ⟵heuristiqueDeChoixDePolarité(x);
25       D_level ⟵ D_level + 1;
26       L_pu ⟵ L_pu ∪ ℓ ;                          /* ℓ est affecté grâce à la PU */
27       si faireReduction() alors reductionClausesApprises(learnt);
```

Remarque 2.6. Une décision (ligne 22 à 26) est effectuée via la propagation unitaire. En effet, un littéral de décision ℓ peut être vu comme l'ajout d'une clause unitaire à la formule Σ : nous avons $\Sigma_{|\ell} = \Sigma \wedge \ell$.

Même si les clauses apprises permettent d'élaguer davantage l'arbre de recherche. L'accroissement de l'ensemble de ces dernières noté `learnt` ralentit considérablement le solveur. En effet, chaque clause apprise en plus augmente la quantité de mémoire nécessaire. De surcroît, comme les clauses apprises sont utilisées dans la propagation unitaire, celle-ci doit les parcourir afin de vérifier quels sont les littéraux à propager. Notons aussi que la propagation unitaire est l'opération la plus coûteuse dans CDCL. En effet, environs 90% du temps CPU lui est dédié. Afin de réduire au maximum ce problème, deux améliorations sont réalisées dans CDCL.

La première part d'un constat simple : nous ne pouvons pas garder la totalité des clauses apprises. Ainsi, une politique de suppression des clauses apprises est mise en place afin de laisser dans le solveur les clauses les plus « utiles » (ligne 27). Néanmoins, il n'est pas facile de juger la qualité d'une clause

apprise et de nombreuses heuristiques sont utilisées dans les solveurs SAT modernes. Nous listons et présentons ces dernières dans la section 2.3.5.

La deuxième amélioration, appelée structure de données paresseuse, est liée à l'algorithme et à la structure de données dédiés à la propagation unitaire. Avant cette amélioration, la propagation unitaire parcourrait la totalité des littéraux d'une clause afin de connaître le littéral à propager quand celui-ci existait. Cette structure de données évite de parcourir tous ces littéraux en marquant uniquement deux littéraux de la clause. Cette structure, décrite plus précisément dans la section 2.3.4, a ainsi énormément augmenté les performances des solveurs SAT actuels.

En plus de ces deux améliorations, les solveurs SAT modernes redémarrent (*restarts*) la recherche suivant une heuristique de redémarrage. En d'autres mots, c'est un *backjump* au niveau de décision zéro, où aucune variable n'a encore été affectée. Cela permet au solveur de réorganiser les valeurs des littéraux. Un mauvais choix de variable en début de recherche peut ainsi être corrigé grâce à un *restart*. Dans la section 2.3.6, nous présenterons plusieurs heuristiques décidant quand ces redémarrages doivent avoir lieu.

Pour terminer la présentation de cet algorithme, n'oublions pas, que la plupart des solveurs SAT prétraitent la formule initiale avant de la résoudre. Dans la section 2.3.7, nous présentons plusieurs techniques de prétraitement.

Dans les prochaines sections, nous allons d'abord décrire quelques heuristiques de choix de variable et de polarité. Par la suite nous présentons les structures de données paresseuses, les politiques de suppression des clauses apprises et les stratégies de redémarrage. Nous clôturons ce chapitre par la description des techniques de prétraitement les plus utilisées.

2.3.3 Choix de variable et choix de polarité

Le choix de la prochaine variable à affecter est certainement le critère le plus important des solveurs SAT. Pour cause, choisir les variables d'un modèle conduit directement à ce modèle. Malheureusement, il est aussi difficile en terme de complexité (*NP-difficile*) de choisir une variable menant à un modèle que d'en trouver un (Liberatore 2000). Pourtant, le choix des variables dans la recherche a un impact considérable sur le nombre d'étapes à réaliser par CDCL, et par conséquent, sur le temps d'exécution (Li et Anbulagan 1997).

Afin de surmonter cette difficulté, plusieurs heuristiques ont été mises en œuvre afin d'estimer la concordance entre un modèle et les variables encore non affectées suivant une interprétation partielle donnée. Pour être efficace, ces heuristiques doivent réduire le plus possible la taille de l'arbre de recherche construit par CDCL. Néanmoins, une heuristique réduisant considérablement la taille de l'arbre de recherche mais gaspillant trop de temps de calcul, peut être moins efficace qu'une heuristique moins coûteuse en temps mais générant plus de nœuds. Une bonne heuristique est donc un bon compromis entre le temps de calcul lui étant dédié et la « qualité » de son choix.

Ces dernières années, diverses heuristiques de branchements ont été proposées. Nous distinguons en général, deux types d'approches :

- les approches *look-back* (rétrospectives) se basent sur les expériences vécues dans le passé par rapport à l'état courant de la recherche. Ces expériences sont les variables affectées, les conflits interceptés et les clauses apprises jusqu'au moment présent. La plupart de ces heuristiques sont donc en adéquation avec DPLL et CDCL ;
- En revanche, les approches *look-ahead* (prospectives) se basent sur des expériences futures par rapport à l'état courant de la recherche. En d'autres mots, l'arbre de recherche est de nouveau

exploré afin de déduire une variable de décision pour le niveau courant de la recherche. Cela implique donc l'affectation d'autres variables, la découverte de nouveaux conflits ainsi que des *backtracks* afin de retourner dans l'état présent de la recherche.

Il est important de noter que les approches *look-back* sont en adéquation avec DPLL et CDCL. En revanche, les heuristiques *look-ahead* sont le fruit d'un nouveau type de solveur basé sur DPLL et encore appelé solveur *look-ahead*. Cette approche est plus efficace sur les instances aléatoires mais aussi sur certaines instances conçues et industrielles possédant une structure particulière. Nous traitons dans cette section les heuristiques de choix de variable. Toutefois, le lecteur intéressé par les solveurs *look-ahead* peut se référer au chapitre cinq du livre de Biere *et al.* (2009).

Les heuristiques *look-ahead*

Pour prendre une décision, un solveur *look-ahead* sélectionne d'abord un sous-ensemble de variables. Ensuite, pour chacune d'entre elles, il l'affecte à vrai, examine les clauses qui sont réduites (par propagation unitaire) mais pas satisfaite et *backtrack*. Un fois cela exécuté, il réitère la même procédure, toujours sur cet ensemble de variables, mais en les affectant à faux. Cette méthode, appelée phase *lookahead*, teste l'affectation de différentes variables, afin d'en choisir une, comme variable de décision. La stratégie par défaut des solveurs *lookahead* est basée sur les clauses qui sont réduites, mais pas satisfaites durant la phase *lookahead*. Ainsi, la plupart de ces solveurs utilisent intensivement la propagation unitaire comme heuristique (Pretolani 1993, Freeman 1995a, Li et Anbulagan 1997). Le lecteur intéressé par ces heuristiques peut consulter Biere *et al.* (2009).

Les auteurs du solveur parallèle CC (pour *Cube and Conquer*) utilisent un solveur *lookahead* afin de générer des sous-problèmes (Heule *et al.* 2012). Dans cet article, deux heuristiques *lookaheads* sont présentées. La première, notée $\mathcal{S}_{cls}(v)$, est la somme des clauses réduites et non satisfaites par v. La deuxième, notée $\mathcal{S}_{var}(v)$ est le nombre de variables assignées (v + nombre de propagations) en affectant v. En général, les solveurs *lookaheads* choisissent la variable représentant le plus grand score calculé par $\mathcal{S}(v) \times \mathcal{S}(\neg v)$. Si plusieurs variables ont le même score, l'ordre de celles-ci est réalisé par $\mathcal{S}(v) + \mathcal{S}(\neg v)$. Pour terminer, l'heuristique de polarité est défini suivant $\mathcal{S}(v)$: si $\mathcal{S}(v) < \mathcal{S}(\neg v)$, v est affecté à `faux`, sinon à `vrai`.

Exemple 2.21. Soit la formule $\Sigma = \{(\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_6) \wedge (\neg x_1 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_5 \vee \neg x_6)\}$. Nous avons $\mathcal{S}_{var}(\neg x_6) = 1$ et $\mathcal{S}_{cls}(\neg x_6) = 2$ car deux clauses sont réduites et une seule variable (x_6) est assignée lorsque x_6 est affectée à faux. Autre exemple, $\mathcal{S}_{var}(\neg x_2) = 4$ et $\mathcal{S}_{cls}(\neg x_2) = 1$ car une seule clause est réduite et quatre variables ($\{x_2, x_1, x_6, x_3\}$) sont assignées lorsque x_2 est affectée à faux.

Dequen et Dubois (2004) propose une heuristique **BSH** (*Backbone Search Heuristic*) fondée sur la notion de *backbone*. Plus précisément, l'heuristique proposée sélectionne les variables ayant le plus de chance d'appartenir au *backbone*.

Définition 2.20 (*Backbone*)**.** Un littéral appartient au ***backbone*** d'une formule si et seulement s'il appartient à tous les modèles de la formule.

Les heuristiques *look-back*

Afin de mieux comprendre les relations entre les heuristiques et les algorithmes, nous divisons ces heuristiques en deux catégories :

- celles crées avant l'apprentissage des clauses dans le solveur GRASP (Marques-Silva et Sakallah 1996), basées sur une interprétation partielle (un état de la recherche) et les clauses de la formule initiale ;
- ceux crées après CDCL l'apprentissage des clauses, tirant profit de l'analyse de conflit.

Remarque 2.7. Certaines de ces heuristiques choisissent aussi la valeur de polarité (`vrai` ou `faux`) en même temps que la variable, nous parlons alors de choix du littéral et d'heuristique de branchement.

Sans l'apprentissage des clauses

Soient ℓ^+ (resp. ℓ^-) le nombre d'occurrences du littéral positif ℓ (resp négatif $\neg\ell$) dans les clauses encore insatisfaites par l'interprétation courante. L'heuristique **DLIS** (resp. **DLCS**) définie dans Silva (1999) établie un score à chaque variable v (représentée par ses littéraux ℓ et $\neg\ell$) tel que $\mathcal{S}_{DLIS}(v) = max(\ell^+, \ell^-)$ (resp. $\mathcal{S}_{DLCS}(v) = \ell^+ + \ell^-$). Le plus grand score sélectionne la variable de décision et la valeur de polarité est `vrai` si $\ell^+ > \ell^-$, `faux` sinon. Ces heuristiques cherchent à satisfaire le plus de clauses possibles sans prendre en compte l'impact de la propagation.

L'heuristique **BOHM**, proposée par Buro et Kleine-Büning (1992), a pour but de satisfaire ou réduire la taille des petites clauses. Elle consiste à choisir la variable qui maximise le vecteur $\mathcal{S}_{BOHM}(v) = (s_1(v), s_2(v), \ldots, s_n(v))$ suivant l'ordre lexicographique (de 1 à n, si $\forall v' \neq v,\ s_1(v) > s_1(v')$, nous choisissons v, sinon, en cas d'égalité nous calculons $s_2(v), \ldots$) où $s_n(v)$ est calculé de la manière suivante :

$$s_n(v) = \alpha \times max(h_n(v), h_n(\neg v)) + \beta \times min(h_n(v), h_n(\neg v))$$

où $h_n(x)$ est le nombre de clauses de taille n contenant le littéral x. Les valeurs de α et β sont choisies d'une manière expérimentale. Les auteurs suggèrent de fixer $\alpha = 1$ et $\beta = 2$. Remarquons que $s_n(v) = \neg s_n(v)$, ainsi la polarité choisie est `vrai` si $h_1(v) + \ldots + h_n(v) > h_1(\neg v) + \ldots + h_n(\neg v)$, sinon `faux` . À l'époque, elle a obtenue de très bons résultats sur les instances aléatoires.

L'heuristique **MOMS** pour *Maximum Occurences in Minimum Size Clauses* proposée par Goldberg (1979) sélectionne elle aussi la variable ayant le plus d'occurrences dans les clauses de plus petites tailles. Une variable v est choisie de manière à maximiser la fonction suivante :

$$\mathcal{S}_{MOMS}(v) = (h^*(v) + h^*(\neg v)) \times 2^k + h^*(v) \times h^*(\neg v)$$

où $h^*(\ell)$ est le nombre d'occurrences du littéral ℓ dans les clauses les plus courtes non satisfaites. La valeur de k, tout comme le fait de décider qu'une clause est courte, est donnée de manière expérimentale. Cette heuristique a été améliorée à plusieurs reprises, en essayant par exemple d'équilibrer les arbres produits par l'affectation d'une variable (Dubois *et al.* 1996).

L'heuristique de branchement **JW** proposée par Jeroslow et Wang (1990) est basée sur le même principe que l'heuristique MOMS. Les auteurs introduisent deux heuristiques, lesquelles sont analysées dans (Hooker et Vinay 1994, Barth 1995), afin de fournir un poids aux variables. Le poids d'un littéral ℓ de Σ est calculé à l'aide de la fonction suivante : $\mathcal{S}_{JW}(\ell) = 2^{-|\alpha_1|} + \ldots + 2^{-|\alpha_n|}$ tel que les clauses α_n sont celles contenant ℓ. La première heuristique (JW-OS) proposée consiste à sélectionner le littéral ℓ qui maximise la fonction $\mathcal{S}_{JW}(\ell)$ et la seconde (JW-TS) consiste à identifier la variable x qui maximise $\mathcal{S}_{JW}(\ell) + \mathcal{S}_{JW}(\neg\ell)$, et a assigner la variable x à `vrai` , si $\mathcal{S}_{JW}(\ell) \geq \mathcal{S}_{JW}(\neg\ell)$, et de l'assigner à `faux` sinon.

Avec l'apprentissage des clauses

L'heuristique **VSIDS** (*Variable State Independent Decading Sum*), proposée par Zhang *et al.* (2001a) dans le solveur CHAFF (Moskewicz *et al.* 2001b) maintient un score pour chaque variable. L'idée basique

est que les variables avec un grand score sont les décisions préférées. Les scores des variables sont stockés dans un tableau utilisé pour trouver la variable de décision. Après l'apprentissage d'une clause, les scores associés à ses variables (alors appelée variables touchées) sont incrémentés. Plus précisément, ces variables touchées sont celles apparaissant dans les résolutions lors de l'analyse de conflit. De plus, tous les 256 conflits, la totalité des scores sont divisées par deux (concept que nous appelons « adoucissement ») puis le tableau est trié en fonction des scores suivant un ordre décroisant. L'ordre des variables est donc uniquement mis à jour tous les 256 conflits. La procédure de décision sélectionne la prochaine variable de décision, en cherchant la première variable non assignée la plus proche dans le tableau, c'est-à-dire, celle avec le plus grand score durant le dernier tri. Notons qu'aujourd'hui, les variables touchées sont celles des clauses apprises mais aussi celles des résolvantes dans l'analyse de conflit.

Si les variables sont adoucies (étape de l'adoucissement vue dans VSIDS) à chaque conflit plutôt que tous les 256 conflits, nous obtenons une variante de VSIDS appelée *normalized VSIDS* (**NVSIDS**). C'est une moyenne glissante exponentielle exprimant à quelle fréquence une variable apparaît dans l'analyse des conflits (Biere 2008a). Soient $\mathcal{S}_{NVSIDS}(v)$ (resp. $\mathcal{S}'_{NVSIDS}(v)$) le score avant (resp. après) l'adoucissement et f un facteur compris entre 0 et 1. À chaque conflit, il est nécessaire de mettre à jour la totalité des variables tel que :

- Si la variable est touchée $\mathcal{S}'_{NVSIDS}(v) = f \times \mathcal{S}_{NVSIDS}(v) + (1 - \mathcal{S}_{NVSIDS}(v))$;
- Sinon, $\mathcal{S}'_{NVSIDS}(v) = f \times \mathcal{S}_{NVSIDS}(v)$.

Aujourd'hui, la plupart des solveurs SAT modernes utilisent une variante de VSIDS très efficace appelée **EVSIDS** (pour *exponential VSIDS*). Celle-ci originellement proposée par les auteurs de MINISAT (Eén et Sörenson 2004), procède à un adoucissement exponentiel uniquement sur les variables touchées tel que :

$$\mathcal{S}'_{EVSIDS}(v) = \mathcal{S}_{EVSIDS}(v) + \left(\frac{1}{f}\right)^{conflits}$$

où $conflits$ est le nombre de conflits depuis le début de la recherche et f un facteur compris entre 0 et 1. Ainsi, plus le nombre de conflits augmente, plus le score (aussi appelé activité) est élevé. De ce fait, les variables des conflits récents sont plus importantes que les autres. En revanche, les variables des anciens conflits se retrouvent, au fur et à mesure des conflits générés, moins importantes. Rappelons que nous appelons ce phénomène l'adoucissement des anciens conflits. Par conséquent, plus le facteur f est élevé, moins l'adoucissement sera fort. Le cas extrême $f = 1$ nous donne :

$$\mathcal{S}'_{EVSIDS}(v) = \mathcal{S}_{EVSIDS}(v) + 1$$

Nous nous retrouvons alors avec un VSIDS de base sans aucun adoucissement. Dans la pratique, le solveur LINGELING utilise une valeur fixe de $f \approx 0,83$ tandis que le solveur GLUCOSE adouci de moins en moins les conflits (f commence d'abord à $0,80$, puis tous les 5000 conflits, f est augmenté de 0.01 jusqu'à atteindre $0,95$). De plus, et encore en pratique, l'efficacité de cette heuristique est aussi apportée par :

- l'utilisation des nombres décimaux au lieu des nombres entiers ;
- une structure de données liée à un algorithme de tri par insertion (appelée arbre bicolore, arbre rouge et noir (en anglais *self-balancing binary search tree*)) permettant de trier l'activité des variables très efficacement (espace linéaire + recherche, insertion et suppression en temps logarithmique) ;
- une variable de décision choisie (celle possédant la plus grande activité) est alors supprimée de l'arbre rouge et noir puis réinsérée lorsqu'elle est touchée.

Pour conclure ce paragraphe, EVSIDS est obligée, à un certain moment, d'adoucir encore plus les scores. Rappelons qu'il s'agit de réduire le score de la totalité des variables. Cela est effectué quand un des scores a une valeur trop élevée (dans GLUCOSE, toutes les variables sont multipliées par 10^{-100} dés que l'une d'entre elles dépasse 10^{100}).

Dans Biere et Fröhlich (2015), les auteurs présentent deux heuristiques se rapprochant, en terme de performance de l'heuristique EVSIDS. L'une d'entre elles, appelée **ACIDS** (pour *Average Conflict-Index Decision Score*) est basée sur une heuristique nommée SUM définie par :

$$\mathcal{S}'_{SUM}(v) = \mathcal{S}_{SUM}(v) + c$$

où c est le nombre de conflits depuis le début de la recherche. SUM ajoute simplement au score d'une variable le nombre c et choisit comme variable de décision celle ayant le score le plus élevé. L'idée principale est qu'une aussi simple heuristique réalise quand même un léger adoucissement. Les auteurs décident donc d'augmenter l'adoucissement, ainsi ACIDS est définie par :

$$\mathcal{S}'_{ACIDS}(v) = \frac{\mathcal{S}_{ACIDS}(v) + c}{2}$$

L'avantage de cette heuristique est clairement sa simplicité mais aussi sa performance. En effet, cette dernière est assez proche de EVSIDS. Nous allons maintenant étudier l'heuristique principale de polarité liée à VSIDS et ses variantes.

Définition 2.21 (Composant d'une formule CNF)**.** Soit Σ une formule. Un ensemble de clauses $\Sigma' \subset \Sigma$ est un composant dans Σ si Σ' ne partage aucune variable avec $\Sigma \setminus \Sigma'$ (Σ privée des clauses Σ').

Biere et Sinz (2006) montrent que, par leur nature, de nombreuses instances du monde réel mettent en évidence une structure de composants. Remarquons que ces composants peuvent aussi être découverts dynamiquement.

Exemple 2.22. Soit la formule $\Sigma = \{(a \vee b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee d \vee e) \wedge (a \vee d \vee \neg e)\}$ contenant un seul composant. Toutefois, après l'affectation de a à faux, cette formule peut-être simplifiée en deux composants : $\{(b \vee c) \wedge (b \vee \neg c)\}$ et $\{(d \vee e) \wedge (d \vee \neg e)\}$.

Expérimentalement, les auteurs Pipatsrisawat et Darwiche (2007) observent que les solveurs SAT effectuaient beaucoup de travail redondant. Contrairement à un simple *backtrack* effectué par DPLL, un « long » *backjump* de CDCL efface légitimement une série d'affectations qui aurait pu être la solution d'un composant indépendant. En effet, quand un « long » *backjump* est réalisé, toutes les affectations entre le niveau du conflit et le niveau de la clause assertive sont effacées. Ainsi, il est possible que CDCL résout chaque composant de multiples fois. Pour éviter cela, les auteurs proposent de sauvegarder la dernière polarité obtenue pendant la recherche dans une interprétation complète notée $\mathcal{P}$, alors appelée ***phase saving***. Ainsi, lorsqu'une nouvelle décision sera prise, la variable se verra attribuée la même polarité que précédemment (choisir x si $x \in \mathcal{P}$, $\neg x$ sinon). L'historique des polarités sauvegarde donc les polarités antérieures des variables de décision choisies mais aussi celles des littéraux visités durant les *backtracks*. De cette manière, l'effort pour satisfaire un sous-problème déjà résolu auparavant sera moins important. Le principal désavantage de cette approche est de ne pas assez diversifier la recherche. Afin de pallier ce problème, Biere (2009) propose d'« oublier » une partie de l'interprétation $\mathcal{P}$.

Plus récemment, deux heuristiques (LRB et CHB) basées sur la notion de moyenne glissante exponentielle représentent le problème de choix de variables comme un problème de Monté Carlo (Liang *et al.* 2016b;a). L'heuristique **LRB** (pour *Learning Rate Branching*) (Liang *et al.* 2016b) est basée sur la notion suivante :

Définition 2.22 (Moyenne glissante exponentielle)**.** La **moyenne glissante exponentielle** (**EMA** pour *exponential moving average*) pour une série temporelle de nombre $\langle g_1, g_2, \dots g_n \rangle$, représentant l'évolution d'une quantité spécifique au cours du temps est calculé par :

$$EMA(\langle g_1, g_2, \dots g_n \rangle) = \alpha(1-\alpha)^{n-1} g_1 + \alpha(1-\alpha)^{n-2} g_2 + \dots + \alpha(1-\alpha)^{n-i} g_i + \dots + \alpha g_n$$

avec $0 < i < n$ et α un paramètre dit « de pas » tel que $0 < \alpha < 1$. Ce dernier permet de contrôler les poids qui différencient les données récentes des anciennes. La moyenne glissante exponentielle peut être calculée incrémentalement par :

$$EMA(\langle g_1, g_2, \dots g_n \rangle) = (1-\alpha) \times EMA(\langle g_1, g_2, \dots g_{n-1} \rangle) + \alpha g_n \text{ et } EMA(\langle\rangle) = 0$$

Exemple 2.23. Soient $\mathcal{S}_1 = \langle 1, 2, 3, 4 \rangle$ et $\mathcal{S}_2 = \langle 5, 4, 3, 2 \rangle$. Via une simple moyenne, nous avons $AVG(\mathcal{S}_1) = 2,5$ et $AVG(\mathcal{S}_2) = 3,5$. Par conséquent, $\mathcal{S}_2$ est le meilleur choix. En revanche, si nous calculons les moyennes glissantes exponentielles avec $\alpha = 0.5$. Nous avons $EMA(\mathcal{S}_1) = 3,0625$ et $EMA(\mathcal{S}_2) = 2,5625$. Dans cette situation, $\mathcal{S}_1$ est préférée à $\mathcal{S}_2$ car les données récentes ont un poids plus élevé que les anciennes.

Afin d'exploiter ce type de moyenne dans une heuristique de choix de variable, les auteurs de Liang *et al.* (2016b) se basent sur les conflits et les clauses raisons. Ainsi, la mesure définie cherche la tendance des variables à produire des clauses apprises. Plus précisément, soit $\mathcal{T}$ l'interval de temps en nombre de conflits entre l'affectation de la variable v jusqu'à son retour parmi les variables non assignées (lors d'un *backjump* ou d'un redémarrage). Le ratio mesurant le taux d'apprentissage d'une variable v pour un interval $\mathcal{T}$, noté LR, est défini suivant :

$$LR(v, \mathcal{T}) = \frac{L(v, \mathcal{T})}{L(\mathcal{T})}$$

où $L(\mathcal{T})$ est le nombre de clauses apprises total durant le temps $\mathcal{T}$ et $L(v, \mathcal{T})$ est le nombre de clauses résolvantes (clauses apprises incluses) contenant la variable v produites dans les analyses de conflits pendant le temps $\mathcal{T}$. Afin d'améliorer ce ratio, les auteurs décident d'inclure dans $LR(v, \mathcal{T})$ les clauses raisons (Définition 2.12) induites par les littéraux composant les clauses apprises en plus des clauses résolvantes des analyses de conflits. Cela est défini par un autre ratio nommé RSR :

$$RSR(v, \mathcal{T}) = \frac{R(v, \mathcal{T})}{L(\mathcal{T})}$$

où $R(v, \mathcal{T})$ est le nombre de clauses raisons (contenants v) des littéraux des clauses apprises durant le temps $\mathcal{T}$. L'heuristique LRB consiste à calculer les moyennes glissantes exponentielles de chaque variables v grâce aux deux ratios tel que :

$$EMA(v) = (1-\alpha) \times EMA(v) + \alpha \times (LR(v, \mathcal{T}) + RSR(v, \mathcal{T}))$$

La variable de décision choisie est alors celle maximisant $EMA(v)$. En pratique, les compteurs $L(\mathcal{T})$, $L(v, \mathcal{T})$ et $R(v, \mathcal{T})$ sont incrémentés suivant leurs définitions respectives pendant l'analyse de conflit tandis que $EMA(v)$ est mis à jour dès que la variable v n'est plus assignée (pendant un redémarrage ou un *backjump*). Le paramètre α commence à $0,4$ puis diminue à chaque conflit de $0,000001$ jusqu'à atteindre une valeur minimale de $0,06$. Cela permet d'oublier de moins en moins les anciennes données (comme le facteur f de VSIDS dans GLUCOSE). De plus, à chaque conflit, les variables n'étant pas assignées diminuent leurs moyennes glissantes exponentielles associées tel que $EMA(v) = EMA(v) \times 0,95$. Cette dernière amélioration permet d'éviter d'explorer trop souvent des variables inactives.

2.3.4 Structure de données paresseuses

La propagation unitaire est l'opération la plus utilisée dans un solveur SAT moderne. En effet, le nombre de propagations unitaires engendré par une décision est si élevé que celle-ci utilise 90% du temps CPU total alloué à la résolution d'une instance. C'est pourquoi de nombreux algorithmes ont tenté avec succès d'améliorer son efficacité. Ces algorithmes doivent, lors de la propagation unitaire, reconnaître au plus vite, les clauses unitaires, insatisfaites ou satisfaites. Pour cela, ils se basent sur des structures de données particulières, contenant le moins de données possible (uniquement celles strictement nécessaires) et par conséquent dites « paresseuses ».

L'idée derrière cette structure, est comme son nom l'indique, d'avoir pour chaque clause, deux variables dites sentinelles (*watchs*) pointant vers deux littéraux de la clause encore non affectés. Ces deux littéraux (*watched two literals*) sont garants du fait que la clause n'est ni unitaire ni falsifiée. En effet, afin de savoir si une clause est unitaire ou non, il suffit de vérifier, qu'un seul de ses littéraux est non affecté et que ses autres littéraux sont affectés à faux, par une interprétation. Une des première structure de données basée sur ce concept a originellement été implémentée dans le solveur SATO (Zhang 1997). Néanmoins, celle-ci présentait l'inconvénient de devoir mettre à jour les littéraux sentinelles lors d'un *backtrack*. Ce problème a été résolu par la méthode apportée par Moskewicz *et al.* (2001a) et implémentée dans le solveur CHAFF. Cette dernière reste incontestablement la plus utilisée aujourd'hui (exploitée par MINISAT et GLUCOSE). Nous allons présenter le déroulement de l'algorithme implémenté dans MINISAT (Eén et Sörenson 2004) et GLUCOSE (Audemard et Simon 2009a).

Lorsque les deux littéraux sentinelle d'une clause ont été désignés pour la surveiller, il est nécessaire de garder cette information en mémoire. Pour cela, chaque littéral d'une formule possède une liste de clauses qu'il doit surveiller. Plus formellement, la liste $W_{\neg\ell}$ est composée des clauses α_n tel que ℓ appartient aux clauses α_n et que ℓ est surveillé. En plus de cela, dans une liste $W_{\neg\ell}$, chaque clause est accompagnée d'un littéral lui appartenant dit bloqué. Ce dernier permet de vérifier plus efficacement si celle-ci est satisfaite.

Exemple 2.24. Soit la clause $\alpha = a \vee b \vee c \vee d$, initialement, les listes représentant les sentinelles sont $W_{\neg a} = \{\alpha\}$, $W_{\neg b} = \{\alpha\}$.

En pratique, à l'initialisation, les sentinelles sont toujours associées aux deux premières positions des clauses. Ainsi, les positions de deux littéraux dans une clause sont parfois échangés. De cette manière, lorsqu'un littéral ℓ est propagé à vrai il suffit de regarder la liste des clauses surveillées par le littéral complémentaire $\neg\ell$ et à chercher un autre « surveillant » dans les positions suivant la deuxième position pour chacunes d'elles. L'algorithme 2.6 est celui implémenté dans MINISAT et GLUCOSE. La notation d'un littéral bloqué d'une clause α inclus dans une liste W est ℓ^{α}_{bloque}. Ce littéral est utilisé d'une manière heuristique afin de vérifier plus rapidement si une clause est satisfaite. Si c'est le cas, son parcours est ainsi évité. De plus, nous notons $\alpha[i]$ le littéral de α à la position i.

Exemple 2.25. Continuons avec l'exemple précédant ($\alpha = a \vee b \vee c \vee d$). Soit la séquence de décision $\mathcal{S}_d = \{\neg d, \neg a, \neg b\}$. Lors de l'affection de $\neg d$, rien ne se passe car $W_{\neg d} = \{\emptyset\}$. En revanche, quand $\neg a$ est affecté, la position des littéraux dans la clause change. Cela est déclenché en parcourant la liste $W_{\neg a}$, nous procédons comme suit :

- a et b sont échangés afin de garder le littéral falsifié (a) en deuxième position (ligne 6 à 8) ;
- Nous cherchons un littéral non affecté à faux à partir de a en allant vers les positions suivantes (de la deuxième à la énième position), nous obtenons c. Ce dernier est alors échangé avec a et les listes de sentinelles sont mises à jour en conséquence (ligne 13 à 18).

Algorithme 2.6 : WatchedTwoLiteralsPropagate

Données : $\mathcal{F}$ une formule et ℓ un littéral à propager

```
1  Début
2    pour chaque clause α dans W_¬ℓ faire
3      ℓ^α_bloque ⟵ α;
4      si I(ℓ_bloque) = vrai alors
5        continue ;                          /* Cette clause est satisfaite */
6      si α[0] = ¬ℓ alors
7        α[0] = α[1];
8        α[1] = ¬ℓ;                  /* Toujours avoir ¬ℓ à la position 1 */
9      si α[0] ≠ ℓ_bloque et I(α[0]) = vrai alors
10       ℓ^α_bloque = α[0];
11       continue ;                          /* Cette clause est satisfaite */
12     pour k allant de 2 à size(α) faire
         // Nous recherchons un nouveau littéral non assigné à faux
13       si I(α[k]) != faux alors
14         α[1] = α[k];
15         α[k] = ¬ℓ;
16         delete(W_¬ℓ, α);
17         add(W_¬α[1], α);
18         continue ;
       // Nous avons potentiellement une clause unitaire
19     ℓ^α_bloque = α[0];
20     si I(α[0]) = faux alors
         // Tous les littéraux sont à faux : c'est un conflit
21       retourner ⊥
22     sinon
         // Seul α[0] est non assigné : c'est une clause unitaire
23       WatchedTwoLiteralsPropagate(F, α[0]);
```

Par la suite, quand $\neg b$ est affecté, nous procédons de la même manière :

- b et c sont échangés afin de toujours garder le littéral falsifié (b) en deuxième position ;
- Nous cherchons un littéral non affecté à faux à partir b, il n'y en a plus. Par conséquent, le littéral $\neg c$ doit être propagé afin de satisfaire la clause (ligne 23).

Nous pouvons observer que, grâce à cette structure, lors de l'affectation d'un littéral, seul un sous-ensemble des clauses contenant une occurrence du complémentaire de ce littéral est parcouru. Pourtant ce traitement partiel est suffisant pour déclencher les propagations unitaires durant le parcours de l'espace de recherche. Ceci explique la rapidité de traitement des affectations, et en particulier de la propagation unitaire. De plus, il est important de noter qu'aucune mise à jour n'est nécessaire lors des retours en arrière. Cependant, cette structure de données possède également quelques inconvénients : les clauses

Décision	Liste de sentinelles	Position des littéraux
$\{\emptyset\}$	$W_{\neg a} = \{\alpha\}$ $W_{\neg b} = \{\alpha\}$	$\underline{a} \vee \underline{b} \vee c \vee d$
$\{\neg d\}$	$W_{\neg a} = \{\alpha\}$ $W_{\neg b} = \{\alpha\}$	$\underline{a} \vee \underline{b} \vee c \vee d$
$\{\neg d, \neg a\}$	$W_{\neg b} = \{\alpha\}$ $W_{\neg c} = \{\alpha\}$	$\underline{b} \vee \underline{c} \vee a \vee d$
$\{\neg d, \neg a, \neg b\}$	$W_{\neg b} = \{\alpha\}$ $W_{\neg c} = \{\alpha\}$	$\underline{c} \vee \underline{b} \vee a \vee d$

TABLE 2.3 – Évolution de la propagation avec une structure de données de deux sentinelles (Algorithme 2.6) sur la clause $\alpha = a \vee b \vee c \vee d$ avec la séquence de décision $\mathcal{S}_d = \{\neg d, \neg a, \neg b\}$.

n'étant pas considérées dans leur intégralité, certains traitements tels que l'utilisation d'heuristiques syntaxiques (DLIS, BOHM, ...) ne sont plus possibles car ils requièrent une connaissance complète de l'instance après les affectations. Toutefois, les heuristiques sémantiques basées sur les conflits peuvent être appliquées avec les structures de données paresseuses, la structure du problème n'ayant pas besoin d'être connue.

Pour information, notons que la notion de « *watched literals* » peut facilement être étendue à d'autres situations où il est nécessaire de vérifier à chaque instant une condition. Par exemple, dans le cadre du solveur parallèle, SYRUP (Audemard et Simon 2014) exploite une structure de données paresseuse d'une seule sentinelle par clause afin de vérifier l'insatisfaisabilité de certaines clauses. De plus, l'utilisation de cette structure n'est pas non plus exclusivement réservée au problème SAT.

2.3.5 Politiques de suppression des clauses apprises

L'analyse de conflits et l'apprentissage augmentent significativement l'efficacité des solveurs SAT modernes. Néanmoins, comme le font justement remarquer Marques-Silva et Sakallah (1996), l'accroissement de la base de clauses apprises ralenti considérablement la propagation unitaire. En effet, plus le nombre de clauses apprises est élevé, plus le nombre de propagations unitaires nécessaires pendant la résolution augmente. Afin d'éviter un tel ralentissement, le solveur SAT moderne doit gérer la base de clauses apprises plus efficacement.

Une solution consiste à améliorer l'algorithme de la propagation unitaire afin que celui-ci puisse traiter une plus grande quantité de clauses. Néanmoins, les techniques utilisant les structures de données paresseuses, décrites dans la section précédente, ont atteint une certaine limite. En conséquence, il est très difficile d'augmenter leurs performances.

Face à ces difficultés, il est devenu nécessaire de supprimer définitivement ou d'écarter pendant un certain laps de temps, certaines clauses apprises. Néanmoins, il est très difficile de savoir quelles sont les clauses à supprimer ou à écarter. Il faut alors juger de la « qualité » des clauses grâce à une heuristique. Cette dernière permet de les ordonner en fonction de leur « utilité » dans un espace ou un sous-espace de recherche donné. De plus, le moment où certaines clauses apprises doivent être éliminées doit aussi être déterminé d'une manière heuristique. Nous allons à présent décrire les différentes politiques de suppression des clauses apprises apparues dans la littérature.

Via leur taille

La majorité des solveurs SAT modernes conservent systématiquement les clauses unitaires et binaires. En effet, celles-ci peuvent être gérées plus facilement. De plus, leurs conservations permettent de contraindre fortement l'espace de recherche et donc d'améliorer la propagation unitaire. Notamment, une clause unitaire appartient au *backbone* de la formule.

Via leur Activité

Dans le solveur MINISAT (Eén et Sörenson 2004), une activité est associée à chaque clause apprise. La notion d'activité est la même que dans VSIDS (section 2.3.3) et est ainsi dynamique : les clauses impliquées dans des conflits récents ont une activité plus élevée que les autres. Une variable $max_{apprises}$ limitant le nombre de clauses apprises est initialement fixée à un tiers du nombre de clauses initiales. Ainsi, cela permet de garder plus de clauses apprises quand le problème est plus grand. Pendant la résolution, si cette limite est dépassée, une fonction de réduction des clauses apprises est appelée afin de supprimer la moitié des clauses apprises en fonction de leurs activités. Pour finir, cette limite $max_{apprises}$ augmente suivant le nombre de conflits grâce à une suite géométrique. Tous les $S_{n+1} = S_n \times 1,5$ (avec $S(0) = 100$) conflits, $max_{apprises} = max_{apprises} * 1,1$ clauses. Ainsi, plus le nombre de conflits est élevé, moins le solveur supprime des clauses apprises.

Via l'heuristique LBD

Au lieu d'utiliser l'activité des clauses apprises, Audemard et Simon (2009b) proposent une nouvelle heuristique, nommée LBD, afin d'attribuer une valeur de qualité aux clauses apprises. Cette mesure correspond au nombre de niveaux différents intervenant dans la génération d'une clause apprise. Plus formellement, la valeur LBD d'une clause se calcule comme suit :

Définition 2.23 (LBD (*Literal Block Distance*))**.** Soient Σ une formule, α une clause et $\mathcal{I}$ une interprétation partielle associant un niveau d'affectation à chaque littéral de la clause α. La valeur **LBD** de α est égale au nombre de niveaux de décisions différents des littéraux appartenant à la clause α.

Exemple 2.26. La clause $x_1^4 \vee x_9^3 \vee x_8^2 \vee x_7^1$ produite dans l'analyse de conflit de l'exemple 2.18 à une valeur LBD de 4.

L'idée apportée par les auteurs Audemard et Simon (2009b) derrière cette heuristique est la suivante. Une décision engendre souvent un grand nombre de propagations, alors appelé *block* de propagations. De ce fait, ajouter des dépendances entre des *blocks* indépendants peut être un moyen de réduire le nombre de décisions. Cela est faisable en ajoutant les plus fortes contraintes (clauses apprises) possibles entre ces *blocks*. Plus précisément, la valeur LBD d'une clause apprise est équivalente au nombre de *blocks* de littéraux propagés. Ainsi, les faibles scores LBDs sont les meilleurs. En effet, une clause avec une valeur LBD de deux, alors appelée clause collée (*glue clause*) permet de coller un littéral ou plusieurs littéraux (suivant la taille de la clause) à un autre *block*. Nous pouvons remarquer que les clauses binaires apprises sont toutes des clauses collées. GLUCOSE garde la totalité des clauses collées durant toute la résolution. La fonction de réduction des clauses apprises supprime la moitié de celles-ci en fonction des valeurs LBDs. Cette fonction est appelée suivant un nombre défini de clauses apprises depuis le début de la recherche. Le moment choisi pour réduire la base de clauses apprises est modifié dynamiquement suivant des données recueillies pendant la recherche afin de s'adapter à l'instance (Audemard et Simon 2016). Pour finir, remarquons que la stratégie de redémarrage utilisée dans GLUCOSE est aussi liée aux valeurs LBDs.

Via l'heuristique PSM

Dans Audemard *et al.* (2011), les auteurs remarquent que certaines clauses apprises supprimées définitivement auraient pu être utiles après leurs suppressions. L'idée est alors de « geler » certaines clauses au lieu de les supprimer définitivement afin de les réactiver plus tard. Pour cela, l'ensemble des clauses apprises est divisé en trois sous-ensembles : celles actives, celles inactives (gelées) et celles supprimées. Ces dernières sont totalement supprimées de la mémoire. En revanche, celles actives et inactives sont gardées en mémoire. De plus, les clauses actives participent à la recherche en les associant à la structure de données paresseuse (*watch*) tandis que celles inactives ne sont pas attachées à cette structure, et ne sont donc pas utilisées dans la recherche. Ainsi, la propagation unitaire est effectuée uniquement avec les clauses initiales et actives. L'avantage est que les clauses peuvent passer d'un état « activé » à « inactivé » et vice-versa pendant la recherche. Cela est réalisé grâce à l'heuristique PSM.

Définition 2.24 (PSM (*Progress Saving Measure*))**.** La valeur **PSM** d'une clause est le nombre de littéraux ayant la même polarité que la *phase saving* apparaissant dans cette clause. Pour rappel, la *phase saving* est un tableau contenant les dernières polarités choisies pendant la recherche par l'heuristique de polarité (section 2.3.3).

Exemple 2.27. Soient la clause $a \vee b \vee c$ et l'interprétation représentant les dernières polarités choisies $\mathcal{I} = \{\neg a, b, c\}$. La valeur PSM de la clause est de 2.

L'idée est que puisque que nous utilisons l'heuristique *phase saving* pour choisir la polarité d'une variable de décision, nous avons donc de forte chance qu'une clause sera assignée via la *phase saving*. Plus précisément, l'intuition de l'heuristique PSM est qu'une clause possédant une valeur PSM élevée possède plus de chance d'être satisfaite, et donc d'être moins utile dans la recherche. En revanche, une petite valeur PSM est plus bénéfique à la recherche. Plus précisément, une clause ayant un PSM de 1 peut probablement propager un littéral pendant la propagation unitaire. Plus encore, une valeur PSM de 0 signifie que la clause associée est potentiellement entrée en conflit et est donc utile. Notons tout de même, que cela reste une heuristique et que ces faits dépendent de l'ordre et du moment des affectations pendant la recherche.

Définition 2.25 (Déviation de la *Phase Saving*)**.** Soit d_t la déviation après t appels à la fonction de réduction des clauses apprises. d_t est définie suivant la distance de Hamming $\mathcal{D}_h$ entre deux *Phase Saving* et le nombre de variables $\mathcal{V}$ tel que :

$$d_t = \frac{\mathcal{D}_h(PhaseSaving_t, PhaseSaving_{t-1})}{\mathcal{V}}$$

Par ailleurs, nous notons d_m le minimum de $\{d_1, d_2, \ldots, d_t\}$.

Comme d_m représente le nombre minimum de variables changeant de polarité durant un certain temps de recherche. Nous pouvons faire des hypothèses sur les littéraux d'une clause en fonction de d_m. Ainsi, pour une clause α de taille $|\alpha|$, si PSM$(\alpha) \geq d_m \times |\alpha|$, alors la clause a des chances de ne pas servir dans la suite de la recherche : elle est alors gelée. Inversement, si PSM$(\alpha) < d_m \times |\alpha|$, elle est activée. De plus les clauses restant gelées ou inutilisées pendant un trop long moment sont supprimées via une heuristique.

Via l'analyse de conflit

L'idée de garder les clauses apprises impliquées récemment dans des conflits a déjà été entreprise via leurs activités. Toutefois, il est possible de pousser cette idée plus loin. Précisément, ces heuristiques

consistent à ne pas supprimer les clauses utilisées dans des conflits venant d'apparaître. Par exemple, le solveur ROKK (Yasumoto et Okugawa 2014) utilise ce concept en protégeant ces clauses dites alors « utiles ».

Oh (2015) expose une différence entre les instances satisfaisables et celles insatisfaisables en fonction des clauses apprises. Dans ce papier, une étude des différents composants d'un solveur SAT actuel est effectuée. Concernant la gestion des clauses apprises, plusieurs expérimentations démontrent une différence notable entre les heuristiques utilisées et la consistance d'une instance. Ne pas supprimer les clauses possédant une valeur LBD inférieure ou égale à 5 (resp. à 1) augmente les performances sur les instances insatisfaisables (resp. à la fois sur celles insatisfaisables et sur celles satisfaisables). Cela est théoriquement dû aux *blocks* de propagations agrandis par les clauses possédant un petit LBD. Ces dernières diminuent le nombre de littéraux à décider pendant la recherche afin d'apporter une preuve de l'insatisfaisabilité plus rapidement. En revanche, garder les clauses d'une taille strictement inférieure à 13, améliore l'efficacité du solveur sur les instances satisfaisables. L'idée de Oh (2015) est alors de trouver un compromis entre ces deux faits. Pour cela, il garde toutes les clauses ayant un LBD d'au plus 3 et celles récemment utilisées dans les analyses de conflits possédant un LBD d'au plus 6. Notons que ce solveur obtient de bons résultats en le combinant avec d'autres optimisations aux niveaux des redémarrages et de l'heuristique de choix de variable EVSIDS.

Le solveur parallèle SYRUP (Audemard et Simon 2014) utilise aussi une heuristique basée sur le nombre de fois qu'une clause entre en conflit pour échanger les clauses apprises entre les solveurs séquentiels. En particulier, il partage les clauses utilisées au moins deux fois dans les analyses de conflits (voir section 4.1.8, page 98).

Via les *backjumps*

Une clause apprise générée permet au solveur d'effectuer un retour en arrière à un certain niveau de l'arbre de recherche : un *backjump*. Dans l'approche apportée par Guo *et al.* (2013), le niveau de ce saut arrière est utilisé pour estimer l'importance d'une clause apprise. L'intuition est qu'une clause est pertinente quand elle permet au solveur d'effectuer un *backjump* au niveau le plus haut possible de l'arbre de recherche (le plus proche de la racine). Pour cela, une heuristique est mise en place en fonction du niveau de *backjump* d'une clause c nommée BTL(c). Remarquons que la valeur BTL(c) est aussi mise à jour lorsque la clause propage un littéral. Dans ce cas, si le niveau lors de la propagation noté i est inférieur à la valeur courante de BTL(c), alors BTL(c) $= i$. Pour finir, notons que les auteurs Guo *et al.* (2013) expérimentent plusieurs stratégies d'élimination des clauses. Par exemple, pour MINISAT, ils suppriment la moitié des clauses apprises en fonction de leurs valeurs BTL lorsque la fonction de réduction est appelée.

2.3.6 Stratégies de redémarrage

Les premiers choix effectués lors d'une recherche sont prépondérants. L'idée est que si la recherche échoue depuis un certain temps (évaluée en nombre de retours arrières) alors il est jugé peu probable que la recherche aboutisse en un temps raisonnable. En effet Gomes *et al.* (2000) ont montré expérimentalement qu'exécuter la même approche sur le même problème mais avec des choix initiaux différents conduit à des temps de résolution totalement hétérogènes. Ces expérimentations ont permis d'identifier un phénomène de longue traînée (*heavy tail*).

Les solveurs SAT actuels redémarrent leur recherche de temps en temps afin d'éviter les problèmes de longue traînées Gomes *et al.* (2000). Un tel redémarrage correspond à un *backjump* au niveau 0.

Supposons que nous ayons une formule Σ et ℓ un littéral de Σ. Puis imaginons que le solveur prend une mauvaise décision dès le début de la recherche (au premier niveau de décision), c'est-à-dire, qu'il décide le littéral ℓ tel que $\Sigma_{|\ell}$ est insatisfaisable tandis que Σ est satisfaisable. Il doit alors prouver que $\Sigma_{|\ell}$ est insatisfaisable avant de rechercher une solution dans $\Sigma_{|\neg\ell}$. En conséquence, il est bénéfique de redémarrer la recherche quand le solveur reste « bloqué » trop longtemps sur $\Sigma_{|\ell}$. Toutefois, les informations telles que les clauses apprises, l'activité des variables et la *phase saving* ne sont pas supprimées. La polarité joue d'ailleurs un rôle important, puisque nous remarquons que dans notre exemple, il ne faut pas forcément suivre la *phase saving* ($\Sigma_{|\ell}$) lors d'un redémarrage. De plus, un redémarrage permet aussi de changer l'ordre des variables en fonction de l'heuristique de choix de variable. Ce comportement s'est également avéré utile sur des formules insatisfaisables, car il permet au solveur de se concentrer sur les parties difficiles de la formule. Nous allons maintenant présenter quelques stratégies de redémarrage.

Séries arithmétiques

Cette première solution consiste simplement à avoir une limite de conflits déclenchant un redémarrage. Cette limite est augmentée par une quantité constante à chaque redémarrage. Ce type de stratégie a été utilisé avec différents paramètres dans zCHAFF (Mahajan *et al.* 2005), BERKMIN(Goldberg et Novikov 2007), SIEGE(Ryan 2004) et EUREKA (Nadel *et al.* 2006).

Séries géométriques

Les premières versions de MINISAT (1.14 et 2.0) sont basées sur une limite multipliée par un facteur constant à chaque redémarrage (Eén et Sörenson 2004). Ce solveur commence par initialiser le premier intervalle de conflits $limite = 100$. Ensuite, lorsque le nombre de conflits atteint la valeur limite, un redémarrage est effectué et la valeur limite est augmentée de 50% ($limite = limite \times 1.5$).

Luby

La stratégie par défaut de MINISAT (version 2.1, Eén et Sörenson (2008)) est devenue un standard et de nombreux autres solveurs l'utilisent (Pipatsrisawat et Darwiche 2007, Huang 2007, Sörensson et Eén 2009, Biere 2009). Celle-ci est basée sur la séquence de Luby : $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \ldots\}$. Dans ce solveur, les intervalles entre les redémarrages (en terme de conflits) sont les nombres de la suite de Luby multipliés par 100. Cette stratégie est dite statique car elle est prédéterminée et ne change pas quelque soit les informations obtenues durant la recherche. Cette séquence est connue pour être logarithmiquement optimale quand, théoriquement, les performances de l'algorithme changent en fonction des variables choisies et que nous ne connaissons pas à l'avance ces variations (Luby *et al.* 1993).

Interne-externe

Une autre variante de série géométrique est appelée la stratégie interne-externe (*inner-outer*, Biere (2008b)). Soit i la taille du premier redémarrage en nombre de conflits initialisant la série. Dans ce schéma, la taille d'un redémarrage en nombre de conflits (la valeur *inner*) est multipliée par un facteur f après chaque redémarrage jusqu'à ce qu'une certaine limite est atteinte (la valeur *outer*). Dans cette dernière situation, la valeur *inner* est réinitialisée à i et la valeur *outer* est multipliée par f. Cela permet d'augmenter la taille des redémarrages petit à petit, c'est-à-dire, moins rapidement que Luby. Néanmoins, cette heuristique engendre quelques redémarrages trop longs à cause de sa nature exponentielle. Cela peut

expliquer pourquoi certaines versions de la stratégie interne-externe résolvent significativement moins d'instances insatisfaisables. Cette heuristique est utilisée dans quelques versions du solveur LINGELING.

Basée sur la taille des clauses apprises

Au lieu d'utiliser le nombre de conflits généré par un solveur, la stratégie de Pipatsrisawat et Darwiche (2009) appelée *width-based policies* encourage le solveur à trouver une preuve par réfutation plus courte en déterminant les moments de redémarrage par rapport aux tailles des clauses apprises entrant en conflits. Pour cela, le solveur maintient une limite de taille w pendant la recherche. Toute clause conflictuelle ayant une taille plus grande que la valeur courante w est prise en considération. D'une manière générale, le solveur redémarre dés qu'au moins n clauses sont conflictuelles depuis le dernier redémarrage. Notons que les clauses conflictuelles sont traitées normalement par le solveur et ne sont donc pas supprimées. La valeur de w peut être soit une constante ou soit une valeur changée dynamiquement suivant quelques critères. Notons aussi que l'absence de clauses conflictuelles de taille strictement supérieure à w ne garantie pas que la taille de la preuve par réfutation générée par le solveur soit supérieure à w. Dans Pipatsrisawat et Darwiche (2009), les auteurs étudient différents critères afin de régler dynamiquement la valeur w.

Redémarrage via l'heuristique LBD

L'idée derrière la stratégie du solveur GLUCOSE est la suivante : afin de produire un maximum de bonnes clauses (en fonction de leurs valeurs LBDs), les redémarrages peuvent avoir lieu quand les dernières clauses apprises produites possèdent des valeurs LBD trop grandes. Pour cela, les auteurs Audemard et Simon (2009a) comparent la moyenne actuelle des valeurs LBDs (définie par x derniers conflits) avec la moyenne globale de toutes les valeurs LBDs (définie par tous les conflits depuis le début de la recherche). Si la moyenne actuelle est sensiblement plus élevée que la moyenne globale, un redémarrage est effectué. Formellement, nous avons :

$$\text{Si } AVG_{actuelle} \times K > AVG_{globale} \text{ alors un redémarrage est déclenché.}$$

Plus la valeur K est élevée, moins les redémarrages sont nombreux. En pratique, afin de calculer la moyenne actuelle, une queue de taille fixe X est utilisée. Ainsi, un redémarrage est effectué à condition que la queue soit pleine, après X conflits. Dans la dernière version (4.1) de GLUCOSE, ces valeurs sont, par défaut $X = 50$ et $K = 0,8$.

Par la suite, dans Audemard et Simon (2012), les auteurs ajoutent la possibilité de reporter certains redémarrages. Un redémarrage est reporté quand le nombre de variables assignées augmente soudainement. L'idée est alors de laisser le solveur résoudre une interprétation partielle pouvant être potentiellement proche d'une solution. Cela est effectué de la même manière que les redémarrages (avec une moyenne actuelle et une globale), en prenant les valeurs des littéraux assignés plutôt que les valeurs LBDs. Récemment, Biere et Frohlich (2015) ont proposés cette stratégie de redémarrage en utilisant une moyenne glissante exponentielle (Définition 2.22) comme pour l'heuristique de choix de variable LRB (Liang *et al.* 2016b).

2.3.7 Prétraitement

Aujourd'hui, les solveurs SAT sont utilisés sur de très grandes formules contenant des millions de variables. Afin d'augmenter les performances, un prétraitement est effectué avant la résolution de l'ins-

tance. Le but des méthodes de prétraitement est alors de simplifier la formule par la réduction du nombre de variables et de clauses inutiles.

Failed Literal Probing

Au sein d'un prétraitement, au niveau de décision zéro, la technique *Failed Literal Probing* consiste à propager le littéral ℓ. Si le résultat est un conflit, alors nous pouvons ajouter la clause unitaire $\neg\ell$ et ℓ est alors appelé le littéral en échec (Berre 2001b, Freeman 1995b). Formellement :

$$\text{Si } \Sigma \wedge \ell \models \bot \text{ alors } \Sigma \models \neg\ell$$

Cela est effectué sur la totalité des littéraux de la formule. Plusieurs améliorations peuvent être ajoutées. Si l'affectation de ℓ à vrai propage le littéral ℓ' à une valeur notée X et que l'affection de ℓ à faux propage encore ℓ' à cette valeur X alors ℓ' peut être affecté à X sans problème. Plus formellement :

$$\text{Si } \left\{ \begin{array}{c} \Sigma \wedge \ell \models \ell' \\ \Sigma \wedge \neg\ell \models \ell' \end{array} \right. \text{ Alors } \Sigma \models \ell'$$

De plus, nous avons aussi :

$$\text{Si } \left\{ \begin{array}{c} \Sigma \wedge \ell \models \ell' \\ \Sigma \wedge \neg\ell \models \neg\ell' \end{array} \right. \text{ Alors } \Sigma \models \ell \Leftrightarrow \ell'$$

Et aussi :

$$\text{Si } \left\{ \begin{array}{c} \Sigma \wedge \ell \models \bot \\ \Sigma \wedge \neg\ell \models \ell' \end{array} \right. \text{ Alors } \Sigma \models \ell'$$

Le sondage des littéraux en échecs (*Failed Literal Probing*) est une technique employée dans certains solveurs *look-aheads* (SATZ,OKSOLVER et MARCH) via l'heuristique de choix de variable. Dans ce cas, ces simplifications peuvent être adaptées suivant le niveau de décision. Dans le chapitre 6, nos contributions utilisent ces notions dans le cadre d'un solveur parallèle.

Bounded Variable Elimination

Comme pour l'algorithme DP, l'idée consiste simplement à éliminer des variables grâce à la $\mathcal{V}$-résolution (Définitions 2.3 et 2.4). Pour rappel, afin d'éliminer une variable x, nous calculons toutes les résolvantes grâce aux clauses contenant cette variable x puis nous supprimons ces dernières. Dans la littérature, plusieurs limitations sont employées afin de ne pas trop augmenter le nombre de clauses. Toutefois, les préprocesseurs les plus courants (SATELITE et NIVER) calculent les résolutions d'une variable uniquement si celles-ci n'augmentent pas la taille de la formule initiale (Eén et Biere 2005, Subbarayan et Pradhan 2005).

Self-Subsuming Resolution

Nous pouvons observer que certaines clauses sont similaires car elles possèdent un motif particulier : une clause c_2 peut partiellement subsumer une clause c_1, sauf pour un littéral ℓ de c_1 qui apparaît avec le signe opposé dans c_2.

Exemple 2.28. Soient $c_1 = \{x \vee a \vee b\}$ et $c_2 = \{\neg x \vee a\}$. La $\mathcal{V}$-résolution sur x produit alors la clause $r = \{a \vee b\}$, qui subsume c_1. Nous pouvons donc remplacer sans problème la clause c_1 par la clause r.

Dans ce cas, nous disons que la clause c_1 est raffermie par la *self-subsuming* en utilisant c_2. Cette simplification est alors appelée la *self-subsuming resolution*. Les auteurs (Eén et Biere 2005) montrent qu'ajouter cette technique à la *Bounded Variable Elimination* améliore significativement l'étape du prétraitement. Aujourd'hui, cette technique est utilisée dans la plupart des solveurs. Notons aussi que des travaux exploitent une condition simple et suffisante afin de détecter, durant l'analyse de conflit, les clauses de la formule qui peuvent être réduites par la technique de *subsumption* (Hamadi *et al.* 2009b).

Asymmetric Literal Addition

Nous pouvons utiliser l'inverse de la *self-subsuming resolution* afin d'ajouter un littéral dans une clause (Heule *et al.* 2010). Cela est très utile à un autre prétraitement (les clauses bloquées).

Exemple 2.29. Soient $c_1 = \{a \vee b\}$ et $c_2 = \{\neg x \vee b\}$. Comme la clause (b) subsume c_1, nous pouvons ajouter sans problème le littéral x à $c_1 = \{x \vee a \vee b\}$

Blocked clauses

Définition 2.26 (Clause bloquée et Littéral bloqué)**.** Un littéral ℓ dans une clause c d'une formule CNF Σ bloque c en fonction de Σ si pour toutes les clauses c' de Σ contenant $\neg l$, les résolvantes $\eta[\ell, c, c']$ sont des tautologies (sont $\top$). ℓ (resp. c) est alors appelé le **littéral bloqué** (resp. la **clause bloquée**). La clause c peut être supprimée de la formule sans aucun problème.

Exemple 2.30. Soit $\Sigma = \{ \underbrace{(a \vee b \vee \underline{\ell})}_{\text{clause bloquée par } \ell} \wedge (\underline{\neg \ell} \vee \neg a \vee c) \wedge (\underline{\neg \ell} \vee \neg b \vee d)\}$. Le littéral ℓ bloque la clause $(a \vee b \vee \ell)$ car les résolvantes $(a \vee b \vee \neg a \vee c) = \top$ et $(a \vee b \vee \neg c \vee d) = \top$ sont des tautologies. Nous pouvons donc supprimer la clause bloquée : $\Sigma = \{\neg \ell \vee \neg a \vee c) \wedge (\neg \ell \vee \neg b \vee d)\}$.

Hidden Tautology Elimination

Les auteurs (Heule *et al.* 2010) appliquent l'*Asymmetric Literal Addition* sur une clause d'une formule. Si la clause est bloquée, elle est éliminée, sinon, la forme originale est gardée (sans l'ajout du littéral de l'*Asymmetric Literal Addition*). Cela est effectué sur toutes les clauses de la formule via des graphes d'implications. Cela est appelé la *Blocked clauses Elimination*. D'autres simplifications sont réalisées via ces graphes comme la *Hidden Literal Elimination* (Heule *et al.* 2011). Le solveur LINGELING (Biere 2010) utilise ces prétraitements. Dans Heule *et al.* (2016), les formules représentant le problème des triplets pythagoriciens booléens contiennent beaucoup de clauses bloquées (plus de 14600 sur certaines formules). Ce prétraitement est alors très efficace sur ce genre de problème.

Variable Elimination by Substitution

De nombreuses instances sont traduites en CNF grâce à la transformation de Tseitin (voir section 1.3.1, page 15). Cette traduction implique qu'un nombre linéaire de variables a été ajouté à la formule. Ces dernières définissent ainsi d'autres variables, plus précisément, elles représentent des portes logiques. Dans cette situation, les variables « en sortie » des portes logiques sont fonctionnellement dépendantes des variables « en entrée ». Par exemple, $\{\ldots \vee (x \vee \neg a \vee \neg b) \wedge (\neg x \vee a) \wedge (\neg x \vee b) \vee \ldots\}$ est une partie d'une formule représentant la porte « AND » tel que $x \Leftrightarrow (a \vee b)$, où x est fonctionnellement dépendante. Ces trois clauses représentent alors la définition de la variable x.

Soit une formule Σ contenant une porte logique « AND » de trois clauses notée G et définissant une variable x. Soit aussi R le reste des clauses contenant x mais ne faisant pas partie des clauses de la porte logique. Soient G_x, $G_{\neg x}$, R_x et $R_{\neg x}$ les ensembles de clauses dans lesquelles, respectivement, les littéraux x,$\neg x$, x et $\neg x$ apparaissent. Nous pouvons alors substituer l'ensemble des clauses $G \cup R$ par les résolvantes de $\eta[x, R_x, G_{\neg x}] \cup \eta[x, R_{\neg x}, G_x]$. Les auteurs (Eén et Biere 2005) montrent que cela diminue toujours le nombre de clauses.

Exemple 2.31. La formule :

$$\Sigma = \{\underbrace{(x \vee c) \wedge (x \vee \neg d)}_{R_x} \wedge \underbrace{(x \vee \neg a \vee \neg b)}_{G_x} \wedge \underbrace{(\neg x \vee a) \wedge (\neg x \vee b)}_{G_{\neg x}} \wedge \underbrace{(\neg x \vee \neg e \vee \neg f)}_{R_{\neg x}}\} \text{ (6 clauses)}$$

peut être remplacée par :

$$\Sigma = \{\underbrace{(c \vee a) \wedge (c \vee b) \wedge (\neg d \vee a) \wedge (\neg d \vee b)}_{\eta[x, R_x, G_{\neg x}]} \wedge \underbrace{(\neg a \vee \neg b \vee \neg e \vee f)}_{\eta[x, R_{\neg x}, G_x]}\} \text{ (5 clauses)}$$

Hyper Résolution

Définition 2.27 (*Hyper-Binary-Resolution* et *Hyper-Unary-Resolution*)**.** Une ***hyper-binary-resolution*** est une étape de résolution qui implique plus de deux clauses. Cette résolution prend en entrée une clause de taille n avec $n \geq 2$ notée $(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n)$ et $n-1$ clauses binaires. Ces dernières sont de la forme $\neg \ell_i \vee \ell'$. Cela produit la nouvelle clause binaire $\ell_n \vee \ell'$. Un cas spécial est l'***hyper-unary-resolution***. Si nous avons $(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n)$ et n clauses binaires de la forme $(\neg \ell_i \vee \ell')$, alors l'*hyper-unary-resolution* produit la clause unitaire (ℓ').

Exemple 2.32. Soit $\Sigma = \{(a \vee b \vee c \vee d) \wedge (h \vee \neg a) \wedge (h \vee \neg c) \wedge (h \vee \neg d)\}$. L'*hyper-binary-resolution* produit la clause $(h \vee b)$.

L'*hyper-binary-resolution* combine plusieurs étapes de résolution en une et peut donc être implémentée ainsi. L'avantage est que cette résolution est plus forte que la propagation unitaire quand ces deux techniques sont répétées jusqu'à l'obtention d'un point fixe. L'*hyper-unary-resolution* est utilisée dans SATELITE. Pour toutes clauses c, ce solveur essaye de trouver des clauses binaires qui produisent une clause unitaire via une seule *hyper-unary-resolution*.

Exemple 2.33. Soit $\Sigma = \{(\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b)\}$. L'*hyper-unary-resolution* produit la clause $(\neg x)$.

Vivification

Une dernière technique, énoncée par Piette *et al.* (2008), consiste à réduire certaines clauses de la base grâce à la propagation unitaire. La réduction d'une clause $\alpha = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$ de Σ est obtenue en appliquant la propagation unitaire sur les littéraux complémentaires de α jusqu'à obtenir l'un des cas suivant :

1. $\Sigma_{|\neg\ell_1 \neg\ell_2 \ldots \neg\ell_i} \models^* \bot$ avec $i < n$. Dans ce cas la clause α peut être remplacée par la clause $\alpha' = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_i)$. En effet, puisque l'interprétation $\{\neg\ell_1, \neg\ell_2, \ldots, \neg\ell_i\}$ falsifie Σ la clause α', elle peut être ajoutée à la formule initiale. Comme α' subsume α, il est possible de remplacer α par α' ;
2. $\Sigma_{|\neg\ell_1 \neg\ell_2 \ldots \neg\ell_i} \models^* \ell_j$ tel que $\ell_j \in \alpha$ et $i < j < n$. Ici, le fait que $\Sigma_{|\neg\ell_1 \neg\ell_2 \ldots \neg\ell_i\}} \models^* \ell_j$ nous permet d'inférer la clause $\alpha' = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_i \vee \ell_j)$ laquelle subsume α. De la même manière que pour le premier cas, la clause α peut être remplacée par α' ;
3. $\Sigma_{|\{\neg\ell_1 \neg\ell_2 \ldots \neg\ell_i\}} \models^* \neg\ell_j$ tel que $\ell_j \in \alpha$ et $i < j \leq n$. Dans ce cas, $\Sigma_{|\neg\ell_1 \neg\ell_2 \ldots \neg\ell_i} \models^* \neg\ell_j$ nous permet d'inférer la clause $\alpha' = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_i \vee \neg\ell_j)$. Puisque, α' self-subsume α il est possible de remplacer la clause α par $\eta[\ell_j, \alpha, \alpha']$.

Conclusion

Ces prétraitements sont intégrés dans le processus de résolution pratique du problème SAT. En effet, à l'heure actuelle tous les solveurs effectuent ce genre de simplification avant de commencer la recherche de solutions. Ces techniques, basées sur l'analyse de la structure de la formule initiale, sont appliquées le plus souvent jusqu'à l'obtention d'un point fixe. Cependant, il est possible qu'en cours de recherche, des littéraux unitaires ou des clauses apprises modifient la base de connaissance de telle sorte que les prétraitements redeviennent à nouveau effectifs. Ainsi, lorsqu'un littéral unitaire est appris ou que la recherche retourne au sommet de l'arbre (lors d'un redémarrage par exemple), toutes les techniques de simplification peuvent de nouveau être appliquées (Eén et Biere 2005) : c'est de l'*inprocessing*.

2.3.8 *Inprocessing*

Dans la littérature, quelques solveurs ont significativement amélioré leurs performances en introduisant des techniques de simplification de la formule durant la recherche CDCL. Par exemple, Sörensson et Biere (2009) proposent une approche qui consiste à continuer d'effectuer des résolutions si celles-ci n'augmentent pas la taille de la clause assertive. Cela permet de supprimer des littéraux redondants dans des clauses apprises directement après leur création. Le solveur LINGELING détecte des littéraux équivalents durant la recherche afin de simplifier la formule (Heule *et al.* 2011). En effet, une variable peut être retirée de la formule quand celle-ci est équivalente à une autre (voir la définition 4.2, page 93). D'autres essaient de découvrir des clauses sous-sommées durant l'analyse de conflits (Han et Somenzi 2009, Hamadi *et al.* 2009b). Cependant, les techniques de simplification des clauses habituellement utilisées dans les prétraitement n'augmentent pas significativement les performances quand elles sont appliquées aux clauses apprises durant la recherche.

Plus récemment, les auteurs Luo *et al.* (2017) ont défini une nouvelle technique *inprocessing* qui permet l'éliminer des littéraux redondants dans les clauses apprises en appliquant la propagation unitaire sur les littéraux des clauses (comme pour la vivification, section 2.3.7, page 56, Piette *et al.* (2008)). Cette technique minimisant les clauses apprises est déclenchée avant que le solveur déclenche un redémarrage et affecte uniquement certaines clauses pendant le processus de recherche. Cependant, la minimisation

des clauses n'est pas réalisée à chaque redémarrage. En fait, le nombre de nouvelles clauses apprises entre deux redémarrages n'est pas assez élevé pour que la propagation unitaire sur ces clauses entraîne un conflit. De plus, les auteurs Luo *et al.* (2017) minimisent uniquement les clauses avec une petite valeur LBD car celles ayant une grande valeur LBD possèdent plus de littéraux à propager lors de la minimisation.

La minimisation de Luo *et al.* (2017) propage donc les littéraux complémentaires d'une clause α un par un afin d'obtenir une clause minimisée α'. Deux cas peuvent survenir.

Si un conflit survient, un graphe d'implication le représente. Une analyse retrace alors ce graphe d'implication à partir de la clause conflictuelle (dérivé par la propagation unitaire) afin de collecter les littéraux complémentaires de la clause α appartenant au dernier UIP et de les ajouter dans α'. Remarquons que, les littéraux du dernier UIP appartiennent forcement à l'ensemble des litéraux complémentaire de α (voir section 2.3.1).

Si aucun conflit n'apparaît, nous ajoutons juste le complémentaire du littéral de la propagation effectuée, c'est-à-dire, celui inclut dans la clause α.

Pour finir, remarquons que les auteurs de Luo *et al.* (2017) testent plusieurs configurations suivant trois paramètres :

- à quel redémarrage faire la minimisation ;
- quelles sont les clauses qui doivent être minimiser ;
- dans quel ordre les littéraux d'une clause doivent être propagés lors de la minimisation.

Les résultats expérimentaux de ces travaux montrent que cette minimisation appelée *Learnt Clause Minimization* est particulièrement efficace. Notamment, les quatre solveurs qui l'implémentent sont les quatre meilleurs solveurs de la compétition SAT de 2017 (*main track*).

2.4 Conclusion

Dans ce chapitre, nous avons présenté le solveur SAT d'aujourd'hui accompagné des techniques qui le compose : l'apprentissage, les heuristiques de choix de variable et de polarité, la structure de données paresseuse, les politiques de suppression des clauses apprises, les stratégies de redémarrage, et les techniques de *preprocessing* et d'*inprocessing*. Nous pouvons alors observer que les bonnes performances des solveurs séquentiels d'aujourd'hui proviennent de la composition de plusieurs techniques, qui rendent le programme séquentiel associé très complexe. L'objectif principal de cette thèse est la résolution de problèmes SAT via des solveurs parallèle. Dans le prochain chapitre, nous allons donc présenter ce qu'est le parallélisme afin de montrer les différentes possibilités qu'il nous offre. Par la suite, nous présentons dans le chapitre 4 les différentes méthodes de résolution parallèle du problème SAT existant dans la littérature.

Chapitre

Le parallélisme

3

Sommaire

3.1 Introduction **60**
3.2 Architectures **62**
3.2.1 Architecture de Von Neumann 62
3.2.2 Registre et mémoire cache 63
3.2.3 Pipeline 64
3.2.4 Architectures parallèles 66
3.2.5 Processeur vectoriel 67
3.2.6 Processeur multi-cœur 69
3.3 Architectures de la mémoire en parallèle **70**
3.3.1 Mémoire partagée 70
3.3.2 Mémoire distribuée 72
3.3.3 Hybridation mémoire partagée/distribuée 72
3.4 Évolution des architectures **73**
3.4.1 Processeur many-cœur 73
3.4.2 Super-ordinateur 73
3.5 Principes de la parallélisation **74**
3.5.1 Notions 74
3.5.2 Limites 75
3.5.3 Accélérations théoriques 79
3.6 Modèles parallèles **82**
3.6.1 Sortes de parallélismes 83
3.6.2 Modèles de programmation parallèle 84
3.7 Conclusion **87**

CE CHAPITRE a pour objectif d'exposer le traitement parallèle en science informatique. Nous analysons le parallélisme en général, les matériels qui le supportent, les concepts qui l'analysent, et les modèles de programmation qui le permettent. D'abord, nous introduisons le parallélisme en précisant son contexte, ses enjeux et ses impacts (section 3.1). Ensuite, nous étudions les architectures matérielles permettant le calcul parallèle (section 3.2) et l'accessibilité des données (section 3.3). Celles-ci mettent en oeuvre différentes notions liées à l'algorithmique parallèle (section 3.5). Enfin, nous présentons divers modèles de programmation parallèle (section 3.6).

Nos contributions présentées dans les chapitres 5, 6 et 7 sont liées à la totalité des notions énoncées dans ce chapitre. Plus précisément, les architectures parallèles présentées sont celles que nous utilisons (section 3.2) puis l'accélération et l'extensibilité d'un programme parallèle sont des notions aidant à examiner les performances de nos solveurs (section 3.5). Notamment, une de nos contributions (D-SYRUP, chapitre 7) est directement liée aux modèles de programmation distribués et aux problèmes d'inter-blocages présentés dans la section 3.6.

3.1 Introduction

D'une manière générale, le parallélisme implique que plusieurs actions soient effectuées en même temps. En informatique, il se définit à différents niveaux et dépend d'un grand nombre de notions et de concepts abordés dans ce chapitre.

Définition 3.1 (Parallélisme)**.** Le **parallélisme** consiste à mettre en œuvre des architectures électroniques permettant de traiter des informations de manière simultanée.

Définition 3.2 (Instruction)**.** Une **instruction** informatique désigne une étape dans un programme informatique. Une instruction dicte à l'ordinateur l'action nécessaire qu'il doit effectuer avant de passer à l'instruction suivante. Un programme informatique est constitué d'une suite d'instructions.

Définitions 3.3 (Calcul séquentiel et Calcul parallèle)**.** Le **calcul séquentiel** est une exécution étape par étape, où chaque instruction est déclenchée lorsque la précédente est terminée, même si deux instructions sont indépendantes. Ce calcul s'oppose au **calcul parallèle**, où plusieurs instructions peuvent être exécutées simultanément.

Aujourd'hui, la plupart des ordinateurs sont parallèles, certains plus que d'autres, à travers différentes techniques, et suivant le nombre d'instructions réalisées simultanément. Pourtant, les premiers ordinateurs basés sur l'architecture de Von Neumann (section 3.2.1) n'étaient pas destinés au parallélisme. L'évolution de l'ordinateur vers une machine parallèle s'est faite progressivement. Vers les années 1970, plusieurs instructions à l'intérieur d'un CPU (pour *central processing unit* : un processeur) peuvent être exécutées « à la volée » simultanément. Cette technique est appelée le parallélisme au niveau des instructions (section 3.2.1) et n'est pas contrôlée par l'utilisateur : c'est le compilateur et le CPU qui décide quand et quelles instructions peuvent être traitées en une seule fois. Le parallélisme est dans un premier temps, déployé et réalisé par les matériels informatiques, sans intervention du programmeur.

En 1965, Gordon Moore a décrit un doublement chaque année du nombre de composants par circuit intégré. Mais en 1975, il réévalua sa prédiction pour énoncer ce que l'on nomme aujourd'hui la loi de Moore (Figure 3.1) : le nombre de transistors des microprocesseurs double tous les deux ans (Moore 1975). En conséquence, les machines électroniques sont devenues de moins en moins coûteuses et de plus en plus puissantes, notamment les CPUs et GPUs (*graphics processing unit*, section 3.2.5). Comme le montre la figure 3.2, la finesse de gravure, reflétant la miniaturisation des transistors, diminue encore aujourd'hui (Intel a annoncé sont passage au 10 nm début 2018 avec les processeur Cannonlake). Cette prouesse technologique est l'un des principaux facteurs maintenant la loi de Moore (une division par deux tous les 5 ans de la finesse de gravure). En comparaison, l'épaisseur d'un cheveu humain est de 100 000 nm et le diamètre d'un atome de silicium est de l'ordre de 0,1 nm. Néanmoins, avec une gravure si fine, le nombre de transistors au sein d'un seul processeur est de plus en plus difficile à augmenter et les processeurs mettent plus de temps à être développés qu'auparavant (Gianoli 2017). La montée en puissance des CPUs vers la fin du 20ème siècle a poussée le matériel informatique à la limite physique de la fabrication de puces.

Même si l'on s'attend à ce que la loi de Moore se maintienne dans un avenir proche, une limite, dictée par les lois physiques existe. Cette limite a déjà eu un impact sur la puissance des ordinateurs. Ainsi, la fréquence d'horloge du processeur reste stable (environ 3,5 GHz) depuis quelques années (Figure 3.3). Afin de surmonter cet obstacle, les constructeurs ont créé la technologie parallèle « multi-cœurs » comme étant un processeur composé d'au moins deux cœurs (ou unités de calcul) gravés au sein de la même puce (section 3.5.2). Ce type d'architecture permet d'augmenter la puissance de calcul sans augmenter la fréquence d'horloge, et donc de réduire la quantité de chaleur dissipée par l'effet Joule. Les processeurs

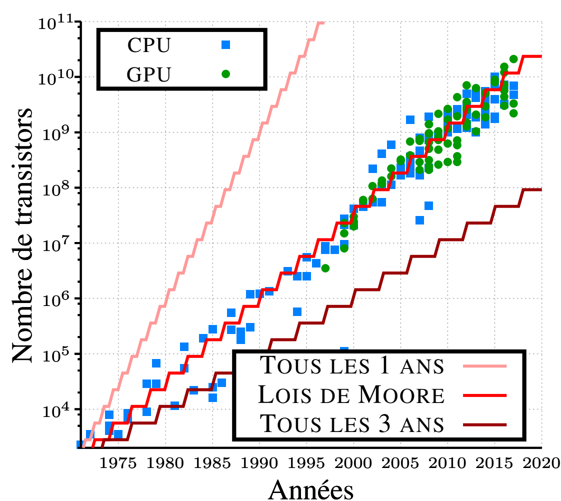


FIGURE 3.1 – Évolution du nombre de transistors (échelle logarithmique) des processeurs commerciaux (CPU et GPU) de 1970 à 2017. La droite rouge correspond à la loi de Moore : un doublement du nombre de transistors tous les deux ans (Source des données : Wikipedia).

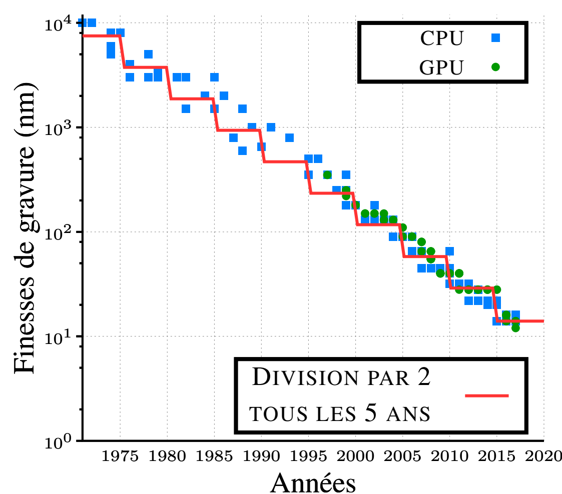


FIGURE 3.2 – Évolution de la finesse de gravure (échelle logarithmique) des processeurs commerciaux (CPU et GPU) de 1970 à 2017. La droite rouge correspond à une division par deux de la finesse de gravure tous les cinq ans (Source des données : Wikipedia).

multi-cœurs sont apparus parce qu'en pratique, l'augmentation de la fréquence devenait onéreuse, compliquée et faisait face à de graves problèmes de refroidissement des circuits. Par conséquent, la solution la plus évidente a été de privilégier non plus la fréquence, mais d'accroître la puissance grâce à une architecture parallèle, de façon à pouvoir augmenter le nombre d'opérations exécutées simultanément en un cycle d'horloge. Les premiers exemplaires de processeurs multi-cœurs d'Intel et d'AMD sont arrivés sur le marché des ordinateurs personnels en 2005. Depuis, le nombre de cœurs au sein d'un même processeur ne cesse d'augmenter (Figure 3.4). Par exemple, les processeurs Intel Xeon Phi sont composés de 57 à 72 cœurs possédant seulement chacun une fréquence d'horloge de 1,1 GHz à 1,5 Ghz.

Malheureusement, les architectures parallèles sont plus compliquées à prendre en main : le programmeur doit apprendre des nouveaux concepts de programmation parallèle afin de profiter pleinement de toute la puissance de la machine. De cet inconvénient, plusieurs modèles de programmation parallèle sont apparus afin de guider les programmeurs. Ces modèles sont constitués de nombreuses bibliothèques utilisant différentes approches (section 3.6). Un autre désavantage, plus important et général, est le fait que les algorithmes séquentiels doivent être totalement repensés. En effet, pour un problème donné, un algorithme parallèle est beaucoup plus compliqué à concevoir, à comprendre et à coder qu'un algorithme séquentiel. Cela est une tâche difficile sachant que le gain espéré n'est pas toujours celui escompté (section 3.5). Un algorithme parallèle exécuté sur un nombre x de cœurs (d'unités de calcul) n'a pas très souvent un temps d'exécution x fois supérieur à sa version séquentielle. Le profit du parallélisme dépend du problème traité et peut parfois même être nul par rapport à un calcul séquentiel, voir négatif dans certaine situation où les ressources doivent être partagées. Cependant,le traitement parallèle comporte plusieurs atouts que ne possèdent pas celui séquentiel.

Premièrement, le calcul parallèle est mieux adapté pour la modélisation, la simulation et la compréhension des phénomènes du monde réel. En effet, le monde autour de nous est massivement parallèle par sa nature, faisant interagir plusieurs entités dans un espace temps. Dans le domaine physique, le massivement parallèle a une place très importante : simulation de la formation de la galaxie, du mouvement

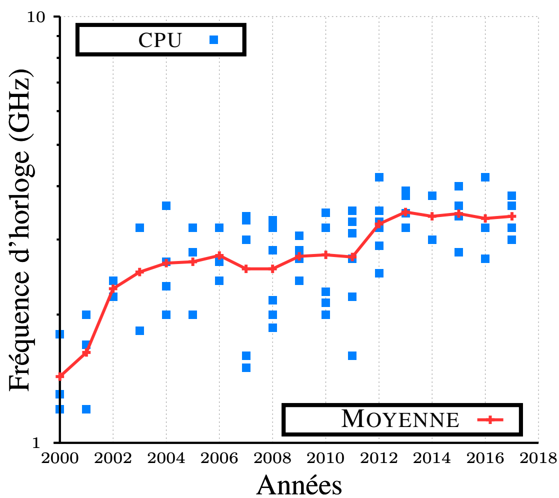


FIGURE 3.3 – Évolution de la fréquence d'horloge (échelle logarithmique) de 2000 à 2017 des processeurs (CPU) commerciaux. La droite rouge correspond à la moyenne par année (Source des données : Wikipedia).

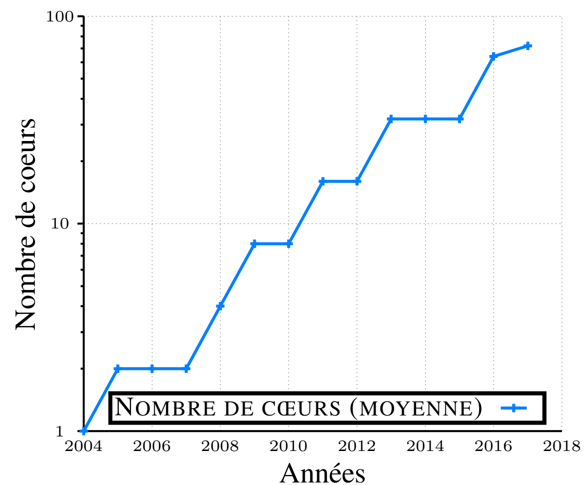


FIGURE 3.4 – Évolution du nombre de cœurs des processeurs (CPU) commerciaux (échelle logarithmique) de 2004 à 2017 (Source des données : Wikipedia).

des planètes, du mouvement des plaques tectoniques, des changements climatiques et de la prévision des tornades (Ugo *et al.* 2015, Keiichiro et Takeshi 2012, Robert et Michael 2016). Dans le domaine industriel, nous pouvons citer l'assemblage des automobiles et avions, ainsi que le trafic autoroutier.

Deuxièmement, le traitement parallèle permet de résoudre des problèmes plus grands et plus complexes comme les *Grand Challenge Problems* (National Science Foundation 2011). Ces problèmes sont fondamentaux et ont été sélectionnés dans le but d'être résolus par des applications informatiques utilisant des ressources matérielles gigantesques hypothétiquement disponibles dans un avenir proche.

Finalement, la parallélisation permet de tirer profit de ressources très éloignées grâce à internet comme le projet SETI@home (Anderson *et al.* 2002). Dans la suite, nous décrivons les architectures parallèles, c'est-à-dire, les technologies matérielles permettant d'exécuter plusieurs actions simultanément.

3.2 Architectures

Afin de comprendre le traitement parallèle, il est nécessaire de connaître les technologies ainsi que l'équipement nécessaire à sa réalisation. Étant la base des architectures parallèles, nous commençons par présenter l'ordinateur de Von Neumann qui permet d'exécuter séquentiellement les instructions. Celui-ci peut alors être défini comme un ordinateur séquentiel.

3.2.1 Architecture de Von Neumann

Même si ils ont en commun d'exécuter les instructions machines des programmes informatiques, les ordinateurs, et plus particulièrement, leurs processeurs, se différencient par un très grand nombre de spécificités (nombre de transistors, types d'architecture, fréquence, mémoire, . . .). Pourtant, avec un très

haut niveau d'abstraction, l'architecture séquentielle de base peut être décrite comme une architecture dite de Von Neumann.

Définitions 3.4 (L'architecture de Von Neumann)**.** L'**architecture de Von Neumann** est principalement composée d'une mémoire indivisible qui stocke à la fois le programme et les données ainsi qu'une unité de traitement qui exécute les instructions en opérant sur les données. Celle-ci décompose l'ordinateur en quatre parties distinctes :

- L'**Unité Arithmétique et Logique** (UAL) effectue les opérations de base.
- L'**Unité de Contrôle** (UC) est chargée du « séquençage » des opérations.
- La **Mémoire** (MEM) contient à la fois les données et le programme qui indiquera à l'unité de contrôle quels sont les calculs à faire sur ces données.
- Les **dispositifs d'Entrée-Sortie** (ES) permettent de communiquer avec le monde extérieur (écran, clavier, souris, . . .).

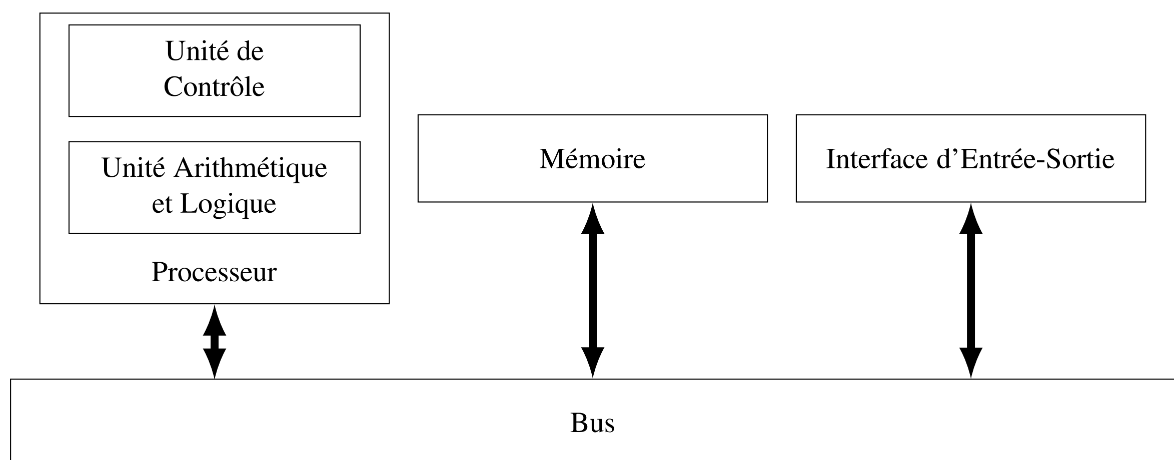


FIGURE 3.5 – Architecture de Von Neumann.

La figure 3.5 présente les relations entre les différents composants (processeur, mémoire principale et entrée-sortie) de l'architecture de Von Neumann. Cette architecture possède un inconvénient encore d'actualité aujourd'hui : l'accès à la mémoire (lecture et écriture) conduit à un goulot d'étranglement car elle possède à la fois les instructions et les données. Le processeur est donc sans cesse obligé d'attendre les données et l'ordinateur se retrouve considérablement ralenti. Autrement dit, l'accès à la mémoire ne suit pas la cadence du processeur. Ce goulot d'étranglement est un problème fondamental en informatique dont la gravité augmente avec chaque nouvelle génération de processeurs. L'architecture de Von Neumann s'est améliorée et sophistiquée jusqu'à atteindre l'architecture d'aujourd'hui et cela a commencé avec plusieurs techniques permettant d'atténuer ce goulot d'étranglement. La plus connue consiste à ajouter des zones mémoires juste à coté du processeur afin d'en accélérer ses accès et ses écritures (registres et mémoires caches).

3.2.2 Registre et mémoire cache

Définition 3.5 (Registres)**.** Les **registres** sont le niveau de mémoire le plus proche du processeur. Il s'agit de la mémoire la plus rapide d'un ordinateur, mais leur capacité dépasse rarement quelques dizaines d'octets car la place dans un microprocesseur est limitée. Les programmes écrits en langage de bas

niveau peuvent directement utiliser ces registres. Certains registres ont certaines fonctionnalités bien précises comme le compteur ordinal qui contient l'adresse de la prochaine instruction à traiter.

Définition 3.6 (Mémoire cache)**.** Une **mémoire cache** ou **antémémoire** est une mémoire qui sert d'intermédiaire entre une autre source de mémoire et le processeur afin de diminuer le temps d'un accès ultérieur d'un processeur à ces données. Il existe différents niveaux de mémoire cache, qui ont une latence plus faible et une bande passante plus élevée que la mémoire principale.

Ces deux sortes de mémoires réduisent considérablement le goulot d'étranglement de l'architecture de Von Neumann. Contrairement à la mémoire cache, les registres sont directement accessibles par le processeur pour les instructions. La mémoire cache, comme son nom l'indique, permet surtout d'avoir un accès plus rapide aux données stockées sur un périphérique plus lent. Autrement dit, si une donnée de la mémoire principale est réutilisée plusieurs fois, elle est mise dans le « cache » afin d'être lue plus rapidement. Plus le processeur est rapide, plus les constructeurs doivent ajouter différents niveaux de mémoire cache afin de ne pas le ralentir. Ainsi, aujourd'hui nous avons trois niveaux de mémoire cache plus ou moins rapide (L1, L2 et L3) en fonction du nombre de cœurs. Remarquons que les mémoires caches L1 et L2 sont locales à un seul cœur tandis que la mémoire L3 est partagée par tous les cœurs d'un processeur.

Aujourd'hui, cette mémoire cache L3 est très utile, particulièrement sur les processeurs multi-cœurs (section 3.2.6). Elle utilise 20 à 25 % de la place dans un microprocesseur. Le Tableau 3.1 présente les différentes sortes de mémoire associées à un processeur récent (2010). Il est intéressant de noter qu'un programme utilisant une quantité de mémoire partagée (entre différentes unités de calcul, voir section 3.2.6) plus grande que le cache L3 peut donc être sévèrement impacté. Dans ce tableau, la latence correspond au retard que peut avoir le processeur en cycle d'horloge à cause d'un accès à une mémoire. Chaque instruction nécessite au moins un cycle d'horloge et ce processeur cadencé à 2,8 GHz possède environs 2800 millions de cycles d'horloges par seconde soit 2,8 cycles d'horloges par nanosecondes.

Mémoire	**Taille** (Bytes)	**Latence** (Cycles d'horloges)	**Temps d'accès** (Nanosecondes)
Registres	4 à 32 Bytes	≤ 1	≤ 1
Cache L1	64 KB par cœurs	~ 4	$\sim 1{,}2$
Cache L2	256 KB par cœurs	~ 10	~ 3
Cache L3	4 MB à 24 MB partagé	$\sim$ 40-75	$\sim$ 12-22
Mémoire principale	1 GB à 16 GB	~ 240	~ 60
Disque dur	1 TB à 8 TB	Non Documenté	~ 4000000

TABLE 3.1 – Hiérarchie de la mémoire d'un Intel Xeon 5500 caractérisé par la vitesse des accès mémoires et leurs tailles.

Le lecteur intéressé peut parcourir le livre de Tanenbaum (2005) afin d'avoir plus de détails concernant les architectures d'ordinateurs. À présent, nous allons présenter une des premières améliorations parallèles de l'architecture de Von Neumann : les pipelines.

3.2.3 Pipeline

Une opération ou une instruction (ADD, SUB, AND, JMP, ...) est composée de plusieurs étapes simples, et pour chaque étape, nous avons un composant dédié dans le processeur. Le pipeline est l'élément d'un processeur dans lequel l'exécution des instructions est découpée en plusieurs étapes. L'idée

derrière les pipelines est la suivante : le processeur peut commencer à exécuter une nouvelle instruction sans attendre que la précédente soit terminée. Ce type de parallélisme est alors appelé *instruction-level parallelism*.

Définitions 3.7 (Parallélisme au niveau des instructions et Pipeline)**.**

Le **parallélisme au niveau des instructions** (ILP pour *instruction-level parallelism*) est une mesure du nombre d'instructions dans un programme informatique pouvant être exécuté simultanément par une seule unité de calcul (par un seul cœur pour un processeur multi-cœurs). Ce parallélisme peut être matériel ou logiciel.

Un **pipeline** ou **chaîne de traitement**, est l'élément d'un processeur dans lequel l'exécution des instructions est découpée en plusieurs étapes. Un pipeline permet le parallélisme au niveau des instructions. La profondeur d'un pipeline représente le nombre de composants nécessaires à l'exécution d'une instruction.

Pour donner un exemple, nous allons aborder l'un des pipelines les plus basiques et connus. Celui-ci s'appelle le pipeline de type RISC classique (*Classic RISC pipeline*) et à été introduit en 1985. Il comporte cinq étapes pour exécuter une instruction représentée par la Table 3.2.

Code	Nom	Description
IF	*Instruction Fetch*	Charge l'instruction à exécuter dans le pipeline
ID	*Instruction Decode*	Décode l'instruction et adresse les registres
EX	*Execute*	Exécute l'instruction (par la ou les unités arithmétiques et logiques)
MEM	*Memory*	Dénote un transfert depuis un registre vers la mémoire ou vice-versa
WB	*Write Back*	Stocke le résultat dans un registre

TABLE 3.2 – Étapes nécessaires afin d'exécuter une instruction par un processeur disposant d'un pipeline de type RISC classique.

Exemple 3.1. Considérons l'instruction triviale ADD(R, S_1, S_2) qui affecte le résultat de l'addition des valeurs des registres S_1 et S_2 dans le registre R. Ici, nous voulons calculer l'opération 1+2=3. Ainsi, cette instruction sera dans la mémoire sous le format *Simple MIPS Instruction* possédant 32 bits que nous pouvons visualiser dans ce tableau :

Opération	$\mathbf{S_1}$	$\mathbf{S_2}$	R	Format	Fonction
000000	00001	00010	00011	00000	100000

Cette instruction est composée de cinq étapes dans un pipeline de type RISC classique :

- IF : Récupère l'instruction à partir de la mémoire.
- ID : Détermine quelle est l'instruction et lit les registres
  - 000000 avec 100000 est l'instruction ADD
  - Les contenus de S_1 et S_2 sont 1 et 2
- EX : Ajoute 1 et 2 = 3
- MEM : Ne fait rien pour cette instruction

Cycle	1	2	3	4	5	6	7	8	9	10
Inst. 1	IF	ID	EX	MEM	WB					
Inst. 2						IF	ID	EX	MEM	WB

TABLE 3.3 – Deux instructions exécutées sur un processeur sans pipeline.

Cycle	1	2	3	4	5	6	7	8	9	10
Inst. 1	IF	ID	EX	MEM	WB					
Inst. 2		IF	ID	EX	MEM	WB				

TABLE 3.4 – Deux instructions exécutées sur un processeur possédant un pipeline de type RISC classique.

- WB : Stocke 3 dans le registre R

Un processeur sans pipeline exécute les instructions les unes aprés les autres (Table 3.3) tandis qu'un processeur possédant un pipeline (Table 3.4) fera le même nombre d'instructions en moins de temps (6 cycles d'horloges contre 10 sans pipeline). Néanmoins, pour que le pipeline puisse traiter deux instructions simultanément, il faut que ces deux instructions soient indépendantes. Il existe plusieurs sortes de dépendances dans un pipeline, voici les plus connues :

- Les dépendances sont structurelles si deux instructions veulent utiliser la même ressource en même temps (unité de calcul, mémoire, registre, composant, ...) ;
- Les dépendances sont de données si deux instructions veulent lire ou écrire dans le même registre ou la même adresse mémoire ;
- Les dépendances de contrôles sont les exceptions matérielles, interruptions et branchements (saut d'une instruction à une autre). Ces derniers posent problème. L'adresse de branchement n'est connue que quelques cycles plus tard vers le décodage de l'instruction dans le meilleur des cas. Et le processeur chargera des instructions inutiles dans son pipeline par erreur.

Aujourd'hui, les constructeurs d'ordinateurs utilise toujours le pipeline afin d'augmenter la puissance des processeurs. Par exemple, le processeur Intel Pentium 4 possède un pipeline d'une profondeur de 20 composants. Certains processeurs récents comme l'Intel Xeon Phi possède un pipeline particulier et adapté aux multi-cœurs. Le pipeline est l'une des premières améliorations parallèles à la machine de Von Neumann. À présent, nous allons introduire et classer d'autres architectures parallèles.

3.2.4 Architectures parallèles

Globalement, les manières permettant de classer le parallélisme sont très nombreuses. Cependant, si nous nous focalisons sur la notion d'architecture parallèle, nous pouvons nous orienter vers deux classifications élémentaires : la taxonomie de Flynn et la classification structurelle basée sur l'organisation de la mémoire (section 3.3).

Flynn (1972) propose une division des architectures en quatre ensembles disjoints basée sur la notion de flux d'instructions et de données. Bien que cette classification soit ancienne, elle permet de donner un aperçu des modes de fonctionnement envisageables pour le parallélisme.

Définitions 3.8 (Taxonomie de Flynn, Flux d'instructions et Flux de données)**.**

Un **flux d'instructions** (resp. **de données**) est un ensemble d'instructions (resp. de données). La **taxonomie de Flynn** classe les architectures en fonction du nombre de flux d'instructions et de données traité simultanément. Comme le montre la Table 3.5, soit il y a un seul flux d'instructions ou de données (*Single*), soit plusieurs (*Multiple*).

		Flux de données	
		Single	*Multiple*
Flux d'instruction	*Single*	SISD	SIMD
	Multiple	MISD	MIMD

TABLE 3.5 – La taxonomie de Flynn

Définitions 3.9 (SISD, SIMD, MISD et MIMD)**.**

- SISD pour ***Single Instruction, Single Data*** : Correspond à un CPU doté d'un seul cœur où une instruction travaille sur une donnée à la fois. Il s'agit du modèle classique d'exécution de Von Neumann.
- SIMD pour ***Single Instruction, Multiple Data*** : Cas des processeurs vectoriels ou unités vectorielles.
- MISD pour ***Multiple Instructions, Single Data*** : Peu répandu car pas naturel, voir inexistant.
- MIMD pour ***Multiple Instructions, Multiple Data*** : Cas des processeurs multi-cœurs avec des processus différents qui travaillent sur des données différentes et des clusters de machines.

Dans la suite de cette section, nous allons présenter les deux architectures parallèles les plus connus à ce jour : les processeurs vectoriels de type SIMD puis les processeurs multi-cœurs de type MIMD.

3.2.5 Processeur vectoriel

Comme le montre la figure 3.6, toutes les unités vectorielles exécutent la même instruction à n'importe quel cycle d'horloge donné. Ces unités de calcul peuvent opérer sur des données différentes.

Exemple 3.2. Un exemple d'application tirant profit de l'architecture SIMD est celle où la même opération doit être effectuée sur un grand nombre de données. Ainsi, pour changer la luminosité d'une image, cette architecture est indispensable. Chaque *pixel* d'une image est la composition de trois valeurs RGB (Rouge, vert et bleu) représentant sa couleur. Pour changer la luminosité, ces trois valeurs doivent d'abord être lues à partir de la mémoire. Ensuite, des additions ou des soustractions sont effectuées sur ces trois valeurs et les résultats sont retournés dans la mémoire. Avec l'architecture SIMD, il y a deux avantages :

- Les données (les valeurs RGB des *pixels*) étant les unes à la suite des autres, sont chargées en une seule fois. Autrement dit, plutôt que d'avoir une longue série d'instructions « retrouve ce *pixel*, ensuite, retrouve le prochain *pixel*, ... », un processeur SIMD fait la même chose en une seule instruction « retrouve n *pixels* ». Avec un processeur traditionnel, cette tâche prendrait beaucoup plus de temps.
- Un autre avantage est que l'instruction de calcul (Addition ou soustraction) est appliquée sur toutes les données chargées en une seule opération.

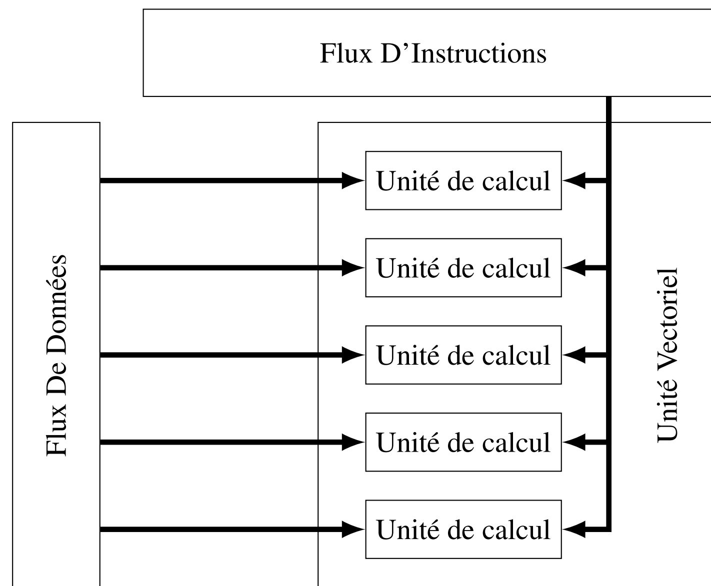


FIGURE 3.6 – Architecture des processeurs vectoriels : SIMD (*Single Instruction, Multiple Data*).

Même si les processeurs vectoriels possèdent l'avantage de pouvoir faires des opérations ou des chargements de plusieurs données en une seule instruction, nous pouvons néanmoins leurs attribuer deux inconvénients.

Premièrement, les problèmes ne possèdent pas tous un algorithme vectoriel efficace. Le champs d'application des processeurs vectoriels reste limités aux simulations physiques, au calcul de matrice et avant tout aux applications graphiques. En effet, il existe des opérations qui ne peuvent pas être exécutées efficacement sur un processeur vectoriel. Par exemple, évaluer un certain nombre de terme de la récurrence $x_{i+1} = ax_i + b_i$ implique plusieurs additions et multiplications, mais alternées (addition, multiplication, additions, multiplication, ...) de sorte qu'il n'est pas possible de faire plusieurs opérations du même type en même temps.

Deuxièmement, le processeur vectoriel dispose de gros fichiers de registre afin de charger plus rapidement les données. Cela augmente la consommation d'énergie et la surface de la puce. De plus, certaines architectures SIMD peuvent avoir des restrictions sur l'alignement des données : celles-ci doivent être les une à la suite des autres. Cela nécessite parfois des copies supplémentaires.

La première utilisation des architectures SIMD sont des super ordinateurs vectoriels au début des années 1970 exécutant une opération sur un vecteur de données en une seule instruction. les applications les plus omniprésentes se retrouvent dans les jeux vidéos : presque toutes les consoles modernes de jeux vidéos depuis 1998 sont composées d'un processeur vectoriel. Aujourd'hui, les processeurs graphiques, ou GPU (*Graphics Processing Unit*), sont des circuits intégrés dans les cartes graphiques (mais pouvant aussi être intégrés sur une carte-mère ou dans un processeur) et assurant les fonctions de calcul de l'affichage. Un processeur graphique contient de nombreuses unités de calcul (plus de 1000 pour la Nvidia Quadro M5000M) permettant ainsi un affichage d'une meilleure qualité. En effet, ces processeurs sont efficaces pour une large palette de tâches graphiques comme le rendu 3D, la gestion de la mémoire vidéo, le traitement du signal vidéo, la décompression vidéo, ... Remarquons qu'aujourd'hui, certains processeurs comme l'Intel Xeon Phi sont une hybridation, c'est-à-dire, ils sont à la fois basés sur l'architectures SIMD et MIMD.

3.2.6 Processeur multi-cœur

Dans l'architecture MIMD représentée par la figure 3.7, chaque unité de calcul peut effectuer simultanément des instructions différentes sur des données différentes. Des instructions différentes ne signifie pas que les processeurs exécutent réellement des programmes différents. La plupart du temps, un seul programme est exécuté et cela est nommé *Single Programme Multiple Data* (SPMD), où le programmeur démarre le même exécutable sur les processeurs parallèles. Contrairement aux processeurs vectoriels (SIMD), étant donné que les différentes instances de l'exécutable peuvent prendre des chemins différents via des déclarations conditionnelles, ou exécuter une quantité différentes d'itérations dans les boucles, ces processeurs ne sont, en général, pas complètement synchronisés. Ce manque de synchronisation est dû au travail effectué par les unités de calcul sur différentes sortes et quantités de données. Cela provoque un déséquilibre des charges accompagné de ralentissements.

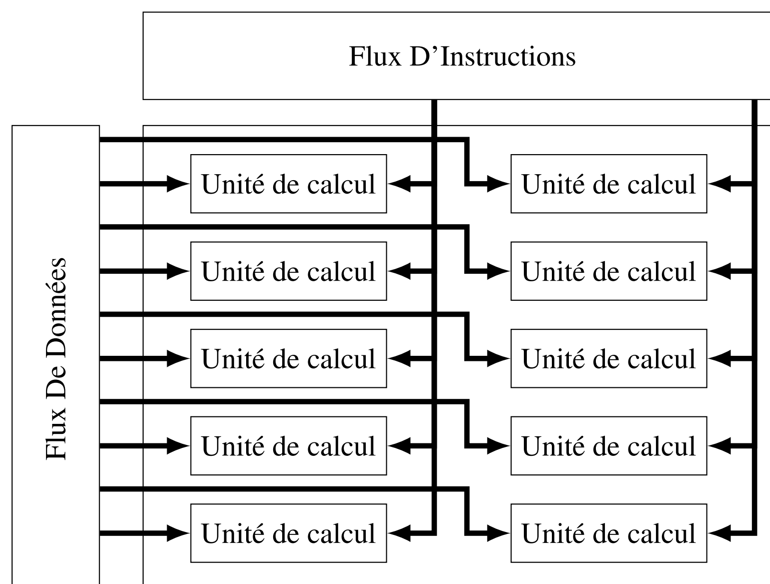


FIGURE 3.7 – Architecture des processeurs multi-cœurs : MIMD (*Multiple Instructions, Multiple Data*).

Définition 3.10 (Cœur de calcul)**.** Un **cœur de calcul** est un ensemble de circuits capables d'exécuter des programmes de façon autonome. Toutes les fonctionnalités nécessaires à l'exécution d'un programme sont présentes dans un cœur : compteur ordinal, registres, unité de calcul, ... Des mémoires caches sont définies pour chaque cœur d'un processeur ou partagées entre eux.

Remarque 3.1. Dans la suite de ce manuscrit, nous employons le terme unité de calcul comme un cœur de calcul.

La figure 3.8 présente la différence entre un ordinateur de Von Neumann (mono-cœur) et un multi-cœurs en fonction de la mémoire cache et du nombre d'unités de calcul (de cœurs). Dans cette figure, les ordinateurs possèdent deux niveaux de mémoire cache (L1 et L2) mais il existe des processeurs composés de trois niveaux de cache (L1, L2 et L3). Dans le cas des processeurs multi-cœurs, le dernier niveau de mémoire cache est souvent partagé entre les cœurs de calcul. Les processeurs multi-cœurs basées sur l'architecture MIMD sont capables de faire du traitement vectoriel mais à plus petite échelle : sur un nombre de données beaucoup plus petit. Cela provient du fait qu'ils possèdent moins d'unités de calcul. C'est aussi pourquoi de nombreuses architectures MIMD incluent des sous composants d'exécutions de type SIMD. À l'heure actuelle, c'est le type d'ordinateur parallèle le plus courant. Les super ordinateurs

les plus modernes entrent dans cette catégorie. Nous allons maintenant présenter la gestion de la mémoire dans de telles architectures.

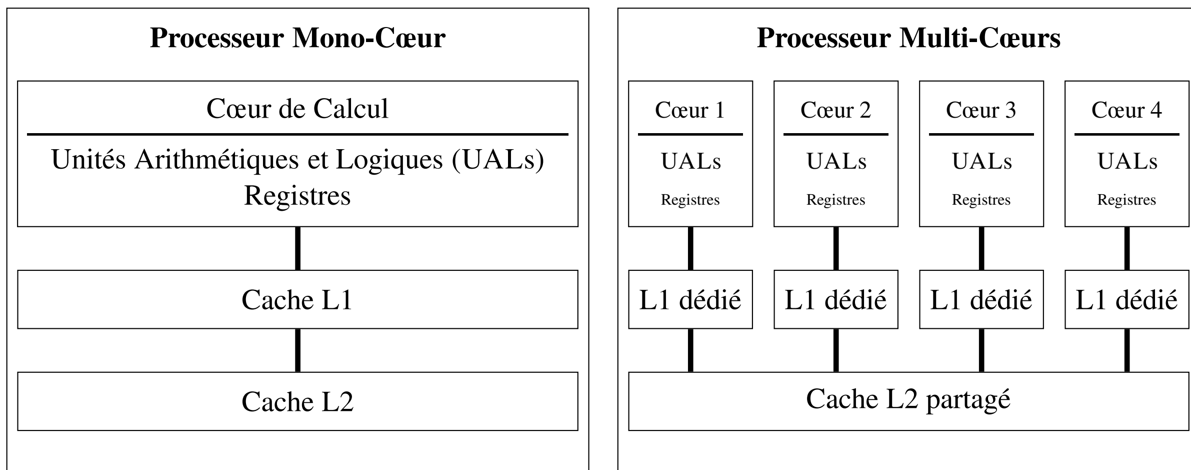


FIGURE 3.8 – Un processeur mono-cœur par rapport à un multi-cœurs.

3.3 Architectures de la mémoire en parallèle

La vitesse des processeurs est sans commune mesure avec la vitesse d'accès à la mémoire. Pour les machines parallèles, où potentiellement plusieurs processeurs veulent accéder à une mémoire au même endroit, ce problème devient encore plus délicat. Nous pouvons caractériser les architectures parallèles par l'approche utilisée pour traiter le problème des accès multiples par plusieurs unités de calcul à un ensemble de données.

3.3.1 Mémoire partagée

Définition 3.11 (Mémoire partagée)**.** La **mémoire partagée** est une mémoire accessible par plusieurs programmes afin qu'ils puissent communiquer entre eux et ainsi éviter les copies redondantes. C'est une manière efficace d'échanger des données entre les programmes.

Les types d'ordinateurs parallèles à mémoire partagée sont nombreux. Généralement, les processeurs ont la possibilité d'accéder à toute la mémoire via un espace d'adressage global.

UMA : *Uniform Memory Access*

Les multiprocesseurs symétriques (SMP) utilisent un seul bus d'échange de données qui représente l'un des premiers styles d'architectures de machines multiprocesseurs, généralement utilisés dans des ordinateurs plus petits jusqu'à huit unités de calcul.

Définition 3.12 (Multiprocesseur symétrique)**.** Un **multiprocesseur symétrique** (SMP pour *Symmetric shared memory MultiProcessor*) est une architecture parallèle qui consiste à multiplier les processeurs identiques au sein d'un ordinateur, de manière à augmenter la puissance de calcul, tout en conservant une unique mémoire partagée.

Définition 3.13 (Architecture de la mémoire de type UMA)**.** Dans une **architecture de type UMA** (pour *Uniform Memory Access*), toutes les unités de calcul partagent une mémoire physique uniforme (Figure 3.9). Le temps d'accès à la mémoire est indépendant de l'unité de calcul faisant la requête ou de la puce contenant les données.

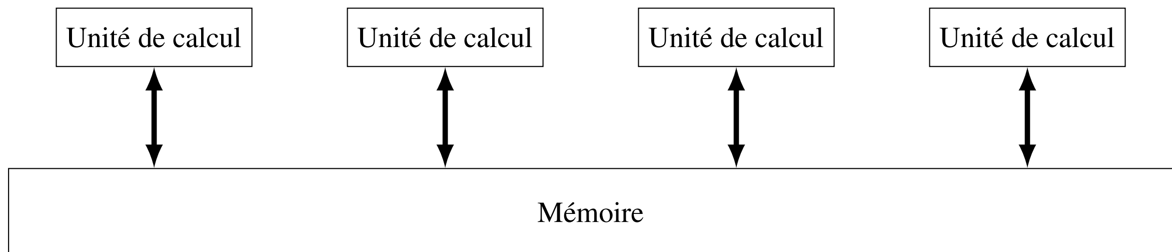


FIGURE 3.9 – Architecture de la mémoire partagée de type UMA : *Uniform Memory Access*.

NUMA : ***Non-Uniform Memory Access***

L'approche de type UMA basée sur la mémoire partagée est évidemment limitée à un petit nombre de processeurs. En effet, à cause d'un encombrement et des accès concurrents par les différents processeurs, l'accès des données via un unique bus, est inefficace. Une architecture plus adaptée devient donc nécessaire pour les systèmes ayant de nombreuses unités de calcul. Comme les bus de données sont extensibles (ils se multiplies facilement), cela semble être un bon choix.

En pratique, les unités de calcul ont une mémoire locale et sont connectées à un réseau. Cela conduit à une situation où une unité de calcul peut accéder à sa propre mémoire, rapidement et à la mémoire d'autres d'unités, plus lentement.

Définition 3.14 (Architecture de la mémoire de type NUMA)**.** Une **architecture de type NUMA** (pour *Non-Uniform Memory Access*) est un système multiprocesseur dans lequel les zones mémoires sont séparées et placées dans des endroits distincts liées à différents bus (Figure 3.10). Vis-à-vis de chaque processeur (ou unité de calcul) et suivant la zone mémoire à accéder, les temps d'accès diffèrent.

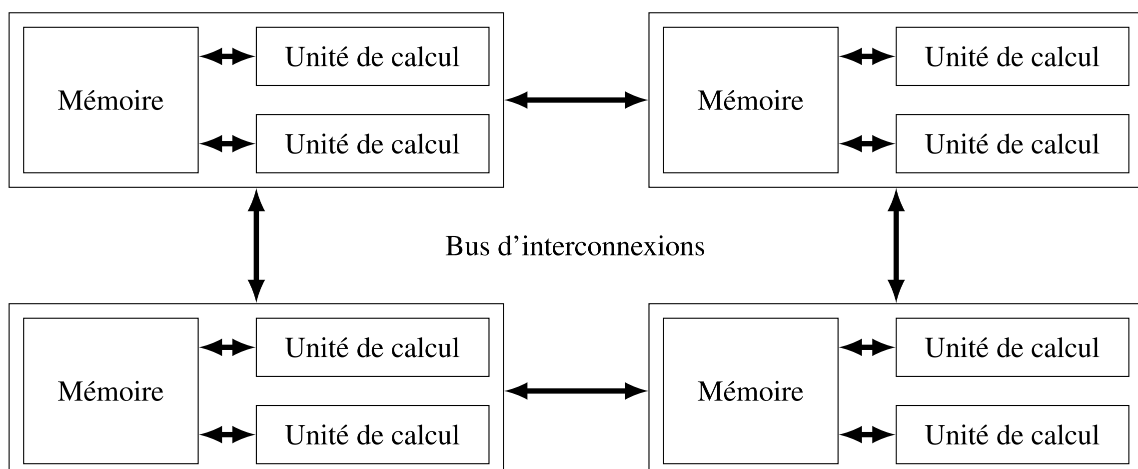


FIGURE 3.10 – Architecture de la mémoire partagée de type NUMA : *Non-Uniform Memory Access*.

Alors que l'approche de type NUMA augmente la rapidité d'accès de certaines données, elle offre des défis pour le système. Imaginez que deux processeurs différents ont chacun une copie d'un emplacement de mémoire dans leur mémoire locale (cache). Si un processeur modifie le contenu de cet emplacement, cette modification doit être propagée aux autres processeurs. Si les deux processeurs essaient de modifier le contenu d'un emplacement de la mémoire en même temps, le comportement du programme peut être indéterminé. Cela pose un problème de cohérence de la mémoire entre les différentes unités de calcul.

Définitions 3.15 (Cohérence du cache et Architecture de la mémoire de type CCNUMA et CCUMA)**.**

Garder des copies de zones mémoires synchronisées est connu comme le problème de la **Cohérence du cache** (en anglais, *cache coherence*) et les multiprocesseurs utilisant ce système sont appelés **CC-NUMA** pour *cache-coherent Non-Uniform Memory Access* et **CCUMA** pour *cache-coherent Uniform Memory Access*.

Comme pour l'architecture UMA, l'accès à la mémoire peut également encombrer le trafic dans le réseau composée de plusieurs bus et la gestion des caches. De plus, le programmeur a la responsabilité de construire des synchronisations qui assure un accès « correct » à la mémoire. En effet, celle-ci ne peut pas être modifiée par plusieurs unités de calcul en même temps (voir section 3.6.2).

3.3.2 Mémoire distribuée

Les systèmes à mémoire distribuée possèdent une caractéristique commune : ils nécessitent un réseau de communications pour connecter toutes les unités de calcul entre elles. Dans cette configuration, les processeurs ont leur propre mémoire locale. Les adresses mémoires d'un processeur ne sont pas connues par les autres processeurs. Par conséquent, il n'y a aucun concept d'espace d'adresse global entre toutes les unités de calcul. Lorsqu'un processeur a besoin d'accéder aux données d'un autre processeur, il appartient habituellement au programmeur de définir explicitement quand et comment les données sont communiquées. La synchronisation entre les programmes est également à la responsabilité du programmeur. Le réseau de base entre les machines est le réseau Ethernet, bien que cela n'est pas toujours le cas (InfiniBand, ...).

Le principal avantage de la mémoire distribuée est sans aucun doute la puissance de calcul. Alors que les architectures sans mémoire distribuée ont un nombre limité d'unités de calcul, celles avec une mémoire distribuée peuvent en rajouter indéfiniment. De plus, l'ajout d'unités de calcul augmente proportionnellement la mémoire disponible. Néanmoins, c'est à la charge du programmeur de régler un grand nombre de détails associé aux communications des données entre les processeurs. De plus, les échanges de données sur le réseau prennent beaucoup plus de temps que ceux de la mémoire partagée.

3.3.3 Hybridation mémoire partagée/distribuée

Aujourd'hui, la plupart des ordinateurs sont une hybridation des deux architectures de la mémoire : partagée et distribuée. C'est notamment le cas des super ordinateurs ou *cluster* de machines. Le composant utilisant la mémoire partagée peut être un CPU ou un GPU. Les tendances actuelles semblent indiquer que ce type d'architecture de mémoire continuera à prévaloir sur les autres. Néanmoins, il est très compliqué pour un programmeur de coder un programme utilisant correctement cette architecture. Les modèles de programmation parallèle pour ce type d'architecture sont nombreux.

3.4 Évolution des architectures

Pour les constructeurs, l'accroissement du nombre de cœurs de calcul est la meilleure manière d'augmenter la puissance d'un processeur. Toutefois, la gestion d'un tel nombre de cœurs demande une métamorphose et l'évolution des architectures actuelles. Ainsi, de nos jours (2017), l'architecture des ordinateurs évolue encore et aura tendance à évoluer de nouveau ces prochaines années. Ainsi, nous pouvons décrire deux architectures spécialement conçues afin d'augmenter la puissance d'un ordinateur : l'une est un processeur nommé « many-cœurs » et l'autre est un ensemble de machines nommé super-ordinateurs.

3.4.1 Processeur many-cœur

Définition 3.16 (Processeurs many-cœurs)**.** Les **Processeurs many-cœurs** sont conçus pour un degré élevé de traitement en parallèle, contenant un grand nombre de cœurs de calcul simples et indépendants. Ils se différencient des multi-cœurs par une architecture mieux adaptée à un très grand nombre de cœurs.

Dans cette sous-section, nous prenons pour exemple le processeur many-cœurs Xeon Phi d'Intel sorti en 2013 et basé sur l'architecture MIC (pour *Many Integrated Core*) élaboré de 2009 à 2013. Ce processeur est un coprocesseur (à utiliser en supplément d'un premier processeur) et est à connecter sur une carte mère en PCI Express. L'accès à la mémoire principale ce fait donc via le bus PCI. Le Xeon Phi possède deux particularités novatrices : une gestion de la mémoire grâce au transfert de celle-ci via un anneau bi-directionnel et la possession de plusieurs unités vectorielles (hybridation des architectures MIMD et SIMD).

Comme le montre la figure 3.11, chaque cœur est conçu pour être efficace et apporte un haut dégré de parallélisme. Ces cœurs possèdent chacun une mémoire cache partagée L2 gardée cohérente grâce à un annuaire d'étiquettes distribuées globalement (TD pour *Tag Directory*). Tous ces composants (cœurs, cache L2, TD et Mémoire) sont directement liés à un anneau bi-directionnel d'interconnexion. De plus, les cœurs possèdent chacun une nouvelle sorte de pipeline permettant de supporter quatre *threads* (différents flux d'instructions) et incluant des composants de type SIMD. Ces derniers apportent l'avantage de pouvoir exécuter deux sortes d'instructions simultanément : ceux provenant des *threads* et ceux provenant d'un calcul vectoriel. L'avantage de cette architecture est donc la rapidité des accès à la mémoire et l'utilisation d'unités de calcul vectorielles. Ainsi, toutes les applications demandant ce genre de ressources seront donc potentiellement plus rapides avec une modification appropriée du code source associé. Les processeurs hybrides (CPU et GPU) de ce genre vont, dans l'avenir, devenir de plus en plus nombreux. Le lecteur intéressé peut consulter le livre de Jeffers et Reinders (2013) expliquant notamment quelles sont les modifications à apporter à un code source afin de tirer pleinement profit d'une telle architecture.

3.4.2 Super-ordinateur

Un superordinateur, ou super calculateur, est un ordinateur conçu pour atteindre les plus hautes performances possibles en ce qui concerne la vitesse de calcul. Ils sont souvent basés sur une architecture de type ccNUMA. Le site `www.top500.org` énumère une liste des 500 super ordinateurs les plus performants. Le « *benchmarck* Linpack » mesure le taux d'exécution en comptant le nombre d'opérations en virgule flottante par seconde (flops) d'un super ordinateur. Cela est déterminé par l'exécution d'un programme qui résout un système dense d'équations linéaires. La barre des 100 petaflops (1 petaflops $= 10^{15}$ flops) a été franchi pour la première fois en 2016 par un superordinateur de nationalité chinoise nommé le « Sunway TaihuLight ».

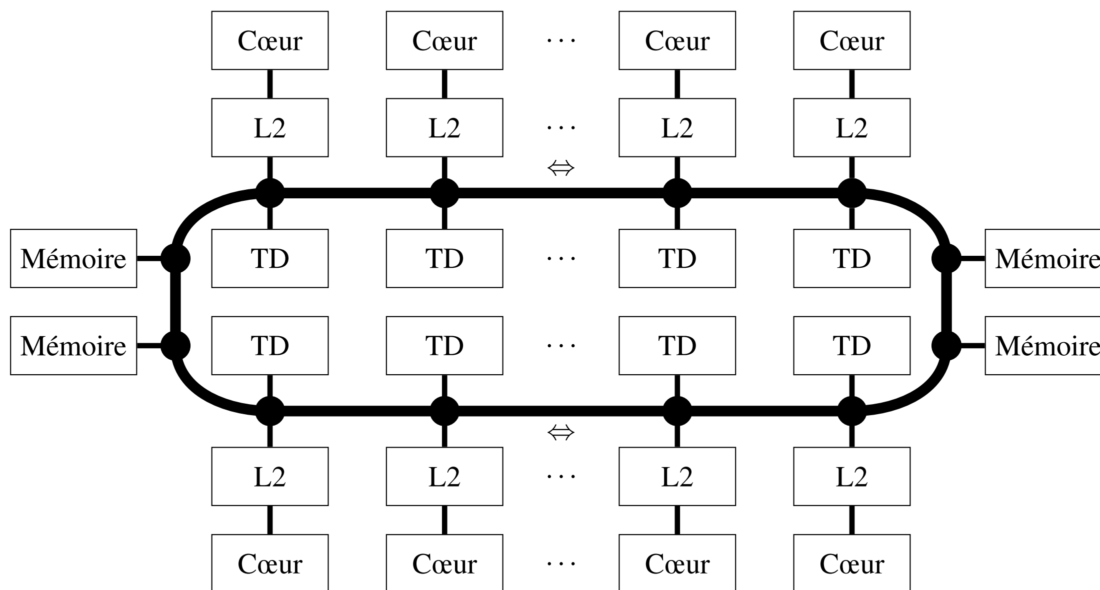


FIGURE 3.11 – Architecture d'un Xeon Phi d'Intel.

3.5 Principes de la parallélisation

Même si il existe de nombreuses manières d'écrire un programme parallèle, elles sont très dépendantes du problème à résoudre. Dans cette section, nous montrons que certains problèmes possèdent même plusieurs façons d'être parallélisés. En effet, les différentes architectures parallèles induisent différentes méthodes de parallélisation. Cette section a pour but de donner une vision globale et d'éclaircir le lecteur se posant la question suivante : comment créer un algorithme parallèle résolvant mon problème efficacement ?

3.5.1 Notions

Avant de discuter concrètement des algorithmes parallèles, nous allons aborder quelques notions très utiles à la compréhension de cette section.

Définitions 3.17 (Algorithme séquentiel et parallèle, exécution séquentielle et parallèle)**.**

Un **algorithme parallèle**, par opposition à un traditionnel **algorithme séquentiel**, est un algorithme qui peut être exécuté en différentes parties sur de nombreux appareils de traitement, puis combiné à la fin pour obtenir le résultat correct.

Une seule unité de calcul exécute une seule tâche à la fois, c'est une **exécution séquentielle**. En revanche, n unités de calcul exécute n tâches distinctes simultanément : c'est une **exécution parallèle**.

Définitions 3.18 (Accélération relative (resp. absolue))**.**

L'**accélération relative** (resp. **absolue**), en anglais *speedup*, notée $S(n)$, représente le nombre de fois que le programme a été accéléré par son exécution parallèle sur n processeurs par rapport à son exécution séquentielle (resp. au meilleur algorithme séquentiel). Elle est souvent calculée suivant les temps d'exécution (séquentiel T_s et parallèle $T_p(n)$) via la formule suivante :

$$S(n) = \frac{T_s}{T_p(n)}$$

Remarque 3.2. De cette manière, quand le temps séquentiel n'est pas disponible ou est indéterminé, nous ne pouvons pas calculer l'accélération.

Remarque 3.3. Il est aussi possible de calculer l'accélération entre deux architectures afin de mesurer, par exemple, l'amélioration apportée par un pipeline.

Exemple 3.3. Dans l'exemple 3.1 de la section 3.2.3, un processeur sans pipeline exécute deux instructions ADD en 10 cycles d'horloges contre 6 avec un pipeline de type RISC. Celui sans pipeline fait donc 5 CPI (cycle par instructions) tandis que celui avec un pipeline fait 3 CPI. L'accélération est alors de :

$$S = \frac{5}{3} = 1,666$$

Définition 3.19 (Efficacité)**.** L'**efficacité** d'un programme parallèle, notée $E(n)$, est l'accélération $S(n)$ divisée par le nombre n d'unités de calcul. Il représente la qualité de la parallélisation par rapport au nombre de processeurs :

$$E(n) = \frac{S(n)}{n}$$

Définitions 3.20 (Accélération linéaire, sous-linéaire et super-linéaire)**.**

Une **accélération linéaire** est obtenue quand l'algorithme parallèle à une accélération $S(n)$ approximativement égale au nombre n d'unités de calcul. Dans ce cas, nous avons $T_s \approx T_p(n) * n$ et les formules suivantes :

$$\textbf{Accélération linéaire} : S(n) = \frac{T_s}{T_p(n)} \approx \frac{T_p(n) * n}{T_p(n)} \approx n \text{ et } E(n) = \frac{S(n)}{n} \approx \frac{n}{n} \approx 1$$

De la même manière, une **accélération sous-linéaire** (resp. **super-linéaire**) est obtenue quand l'algorithme parallèle à une accélération $S(n)$ strictement inférieure (resp. strictement supérieure) au nombre n d'unités de calcul. Nous avons donc les deux formules suivantes :

$$\textbf{Accélération sous-linéaire} : S(n) < n \text{ et } E(n) < 1$$

$$\textbf{Accélération super-linéaire} : S(n) > n \text{ et } E(n) > 1$$

Définition 3.21 (Extensibilité)**.** L'**extensibilité** (en anglais, *scalability*) d'un algorithme parallèle est sa capacité à maintenir son efficacité quelque soit le nombre d'unités de calcul utilisé.

La figure 3.12 expose plusieurs extensibilités possibles représentant différentes accélérations. Elle montre aussi qu'une accélération super-linéaire avec un certain nombre d'unités de calcul peut devenir sous-linéaire avec un plus grand nombre d'unités de calcul. La prochaine section présente les raisons d'une parallélisation qui n'est pas extensible.

3.5.2 Limites

Par définition, un algorithme parallèle va donc se diviser en plusieurs tâches. Plus précisément, un programme parallèle est composé de tâches exécutées sur plusieurs unités de calcul. Supposons que nous avons réussi à diviser un algorithme séquentiel en un nombre de tâches indépendantes et équivalentes en

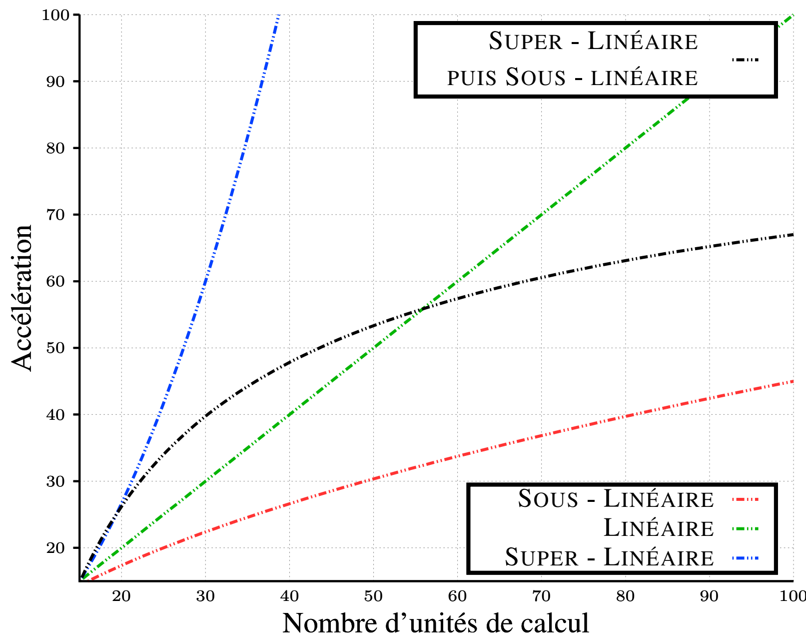


FIGURE 3.12 – Exemple de quatre extensibilités possibles en fonction du nombre d'unités de calcul représentant plusieurs sortes d'accélérations possibles.

terme de quantité de temps (Figure 3.13). De plus, supposons que toutes les unités de calcul exécutent un programme avec exactement la même quantité de temps. Avec de telles conditions, un problème résolu séquentiellement en T_s secondes devrait alors idéalement prendre seulement $T_p = \frac{T_s}{n}$ secondes avec n unités de calcul. Dans cette situation, nous avons donc une accélération linéaire. Ainsi les neufs tâches exécutées avec une unité de calcul dans la figure 3.13 vont pouvoir l'être trois fois plus vite avec trois unités de calcul par un algorithme parallèle (Figure 3.14).

Il ne faut pas, par erreur, faire ces hypothèses. Malheureusement, elles ne sont presque jamais validées. Quelque soit l'algorithme parallèle choisi, cette image parfaite du parallèle n'arrivera probablement jamais. À présent, nous présentons les principales raisons qui font obstacle au parallélisme : l'équilibrage des charges, les dépendances et les communications.

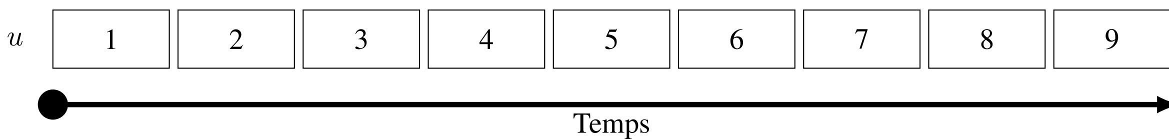


FIGURE 3.13 – Exécution d'un algorithme séquentiel composé de neuf tâches par une seule unité de calcul (u).

Équilibrage de charge

Définition 3.22 (Problème de l'équilibrage de charge)**.** À quelques rares exceptions prés, le problème que l'on doit résoudre, ne peut pas être divisé en morceaux de même complexité. De ce fait, toutes les unités de calcul ne peuvent pas exécuter leurs tâches dans le même délai. Par conséquent, il y a des instants où quelques unités de calcul doivent attendre que d'autres finissent leurs tâches (Figure 3.15). Cela induit que par moments, des ressources ne sont pas pleinement exploitées, à tel point que le temps

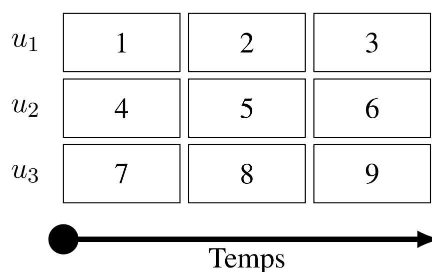


FIGURE 3.14 – Exécution d'un algorithme parallèle en utilisant trois unités de calcul (u_1,u_2 et u_3) avec une accélération linéaire.

nécessaire à l'exécution de l'algorithme augmente. Ce problème, appelé l'**équilibrage de charge** est très fréquent en parallèle.

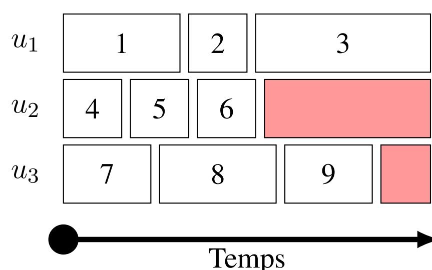


FIGURE 3.15 – Exécution d'un algorithme parallèle présentant le problème d'équilibrage de charge. Les parties rouges montrent les ressources inutilisées induisant une augmentation du temps nécessaire à l'exécution de l'algorithme.

Toutefois, il existe plusieurs techniques pour diminuer au mieux son impact. En outre, l'utilisation de plusieurs machines différentes (par exemple, possédant différents processeurs) est très déconseillé car une telle configuration augmente le déséquilibre des charges. Nous exposons dans cette section les deux principales manières permettant d'atténuer ce problème.

La première idée logique est d'essayer de diviser le problème en plusieurs tâches similaires ayant la même complexité. Néanmoins, il n'est pas souvent possible d'avoir un tel contrôle sur la division d'un problème. Afin d'avoir des tâches les plus équivalentes possibles, une astuce est alors d'entreprendre une division formant de nombreuses petites tâches. Comme le montre la figure 3.16, un grand nombre de petites tâches peut diminuer un déséquilibre antérieur. Malgré tout, ce grand nombre de tâches engendre plus de communications. De surcroît, le surcoût généré par une telle division est parfois trop important. Par exemple, créer de très petites tâches pour le problème SAT, revient à le résoudre séquentiellement. C'est pourquoi cette solution est limitée à certains problèmes, ou certains algorithmes. Nous pouvons citer un grand nombre de problèmes où les tâches sont indépendantes entre elles. Les plus communs sont les applications à destination des GPU. À titre d'exemple, citons la fractale de l'ensemble de Mandelbrot, les applications graphiques où chaque *pixel* est indépendant, la transformation de Fourier discrète, ... Nous discutons de l'équilibrage de charge pour le problème SAT dans le chapitre 4.

Définition 3.23 (Problème parfaitement parallèle (*embarrassingly parallel problem*))**.** Dans le calcul parallèle, un **problème parfaitement parallèle** (en anglais, ***embarrassingly parallel problem***) est celui où il faut peu ou aucun effort pour séparer le problème en un certain nombre de tâches parallèles. Il a donc pour avantage, de ne pas être, ou être très peu concerné par l'équilibrage de charge.

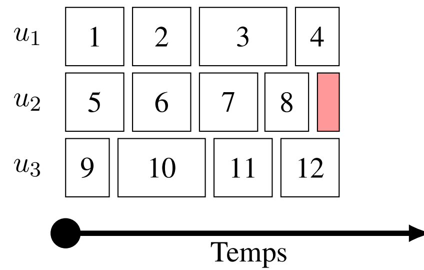


FIGURE 3.16 – Exécution d'un algorithme parallèle présentant le problème d'équilibrage de charge. Il induit un très grand nombre de petites tâches afin de diminuer un potentiel déséquilibre des charges. Les parties rouge montrent les ressources inutilisées induisant une augmentation du temps nécessaire à l'exécution de l'algorithme.

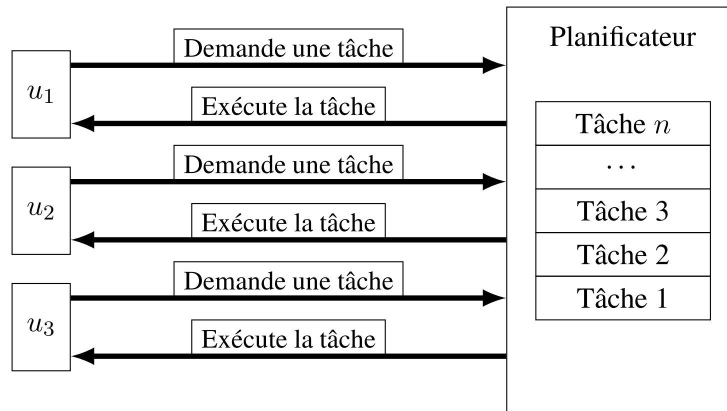


FIGURE 3.17 – Gestion dynamique de l'attribution des tâches aux unités de calcul (u_1,u_2 et u_3) afin de diminuer un déséquilibre antérieur des charges.

La deuxième idée est de gérer dynamiquement l'attribution des tâches aux unités de calcul. Cela a pour effet de diminuer le déséquilibre des charges, pas parfaitement, mais raisonnablement. Cette idée, basée sur un modèle maître/esclave est représenté par la Figure 3.17, impose l'ajout d'une tâche supplémentaire responsable de la planification des autres tâches. Cette tâche est spéciale car elle dure la totalité de l'exécution du programme. De ce fait, il faut lui dédier une unité de calcul ou un *thread* (voir section 3.6.2). Dès qu'une unité de calcul (esclave) termine une tâche, elle en demande une autre au planificateur (maître).

Dépendances entre tâches

Quand nous divisons un problème en tâches, il arrive souvent que certaines tâches doivent être exécutées avant d'autres. Par exemple, il pourrait y avoir des ressources partagées comme des valeurs (ou autres choses) qui ne doivent être calculées qu'une seule fois, mais qui sont nécessaires à d'autres tâches. Cela a pour effet de rendre séquentielle une partie de l'exécution entraînant une augmentation du temps alloué par l'algorithme parallèle (Figure 3.18).

Communications

Pour clore les limitations du parallélisme, nous allons discuter des communications. Un algorithme parallèle peut échanger des données entre les unités de calcul. Cet échange ajoute un coût supplémen-

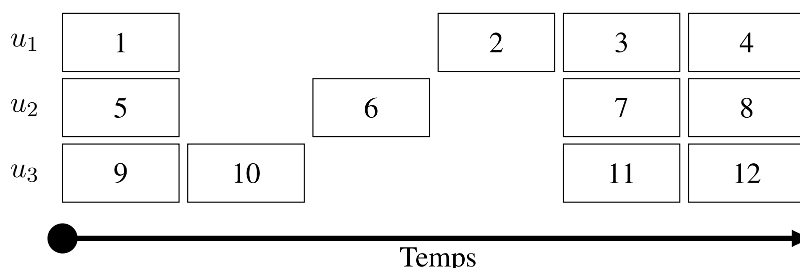


FIGURE 3.18 – Exécution d'un algorithme parallèle présentant un ralentissement dû aux dépendances entre les tâches. La sixième tâche dépend de la dixième et celle-ci de la deuxième. Pour finir, la troisième, septième et onzième tâches dépendent de la deuxième tâche.

taire qui n'était pas présent dans l'algorithme séquentiel (Figure 3.19). Ce surcoût provient de l'ajout de certaines fonctions exécutées sérialement dans l'algorithme parallèle. Certains problèmes ont besoin d'échanger des données afin de pouvoir exécuter les différentes tâches. Dans le chapitre 7, une de nos contributions consiste à améliorer un algorithme parallèle pour le problème SAT en modifiant les moments et les manières de communiquer.

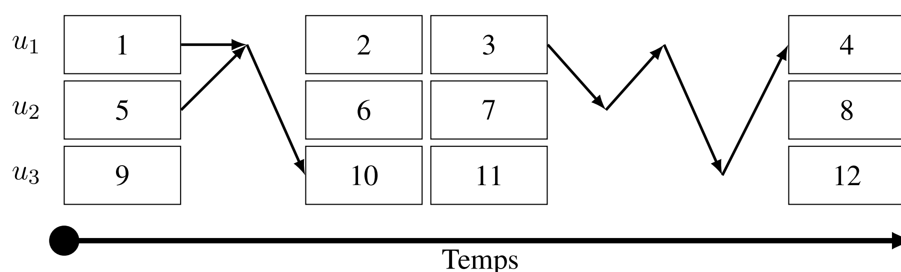


FIGURE 3.19 – Exécution d'un algorithme parallèle présentant un ralentissement dû aux communications.

3.5.3 Accélérations théoriques

Nous pouvons trouver dans la littérature de nombreuses manières permettant de présager l'accélération engendrée par un algorithme parallèle (Amdahl 1967, Gustafson 1988). Dans cette section, nous présentons les deux formules les plus utilisées pour prédire une accélération théorique utilisant plusieurs unités de calcul (multi-cœurs, multiprocesseurs, GPU, ...) : la loi d'Amdahl et la loi de Gustafson. Nous commençons par exposer deux définitions qui viennent compléter celle de l'accélération relative (définition 3.18) présentée au début de cette section.

Remarque 3.4. La charge C d'exécution d'une tâche est sa quantité de travail en entrée et peut augmenter suivant le nombre de données qui doit être traité par un algorithme.

Définition 3.24 (Latence)**.** La **latence** d'une architecture (unité de calcul,...) ou d'un algorithme (séquentiel ou parallèle) est définie via un temps T et une charge C d'exécution d'une tâche par la formule suivante :

$$L = \frac{T}{C} = \frac{1}{V}$$

C'est aussi l'inverse de la vitesse V d'exécution d'une tâche.

Définition 3.25 (Accélération en latence)**.** L'**accélération en latence** entre deux architectures ou algorithmes est définie par la formule suivante :

$$S_{latence} = \frac{L_1}{L_2} = \frac{T_1 * C_2}{T_2 * C_1}$$

Remarque 3.5. Quand C_1 et C_2 sont égaux, on retrouve la définition de l'accélération relative $S(n) = \frac{T_s}{T_p(n)}$ par rapport à un nombre d'unités de calcul n et deux temps d'exécutions différents (un séquentiel T_s et un parallèle T_p) ayant la même charge de travail (résolvant le même problème).

L'accélération en latence est prise pour base par les lois d'Amdahl de Gustafson. Le premier suppose que les charges restent identiques, c'est-à-dire, que la taille des problèmes est identique. De plus, il suppose aussi que la partie parallèle du programme va n fois plus vite, n étant le nombre d'unités de calcul. En revanche, le deuxième suppose que le temps d'exécution des tâches reste égal mais que la tailles des problèmes augmente linéairement en fonction du nombre d'unité de calcul. Il s'agit là de savoir si vous préférez un verre à moitié plein ou à moitié vide, c'est juste deux manières très différentes mais logiques et orthogonales de présager l'accélération. L'une par rapport au temps réel d'exécution et l'autre par rapport à la taille du problème à résoudre. Quoi qu'il en soit, les deux apportent des informations utiles et fondent leurs théories à partir d'une idée commune : un problème (ou un algorithme) comporte une partie non parallèle (séquentielle) et une partie parallèle.

Loi d'Amdahl

Soit F_p la fraction du temps d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle à la suite d'une augmentation du nombre d'unités de calcul. La fraction représentant la partie ne bénéficiant pas de cette amélioration, celle séquentielle, est donc $1 - F_p$. Ainsi, le temps total du programme séquentiel T_s (avec une unité de calcul) est :

$$T_s = (1 - F_p) * T_s + F_p * T_s$$

La loi d'Amdahl suppose que le temps de la partie parallèle est amélioré linéairement en fonction du nombre d'unités de calcul n. Le temps total du programme parallèle T_p est donc :

$$T_p = (1 - F_p) * T_s + \frac{F_p}{n} * T_s$$

Pour finir, nous appliquons l'accélération en latence $S_{latence}$ avec une charge de travail C fixe :

$$S_{latence} = \frac{T_S * C}{T_p * C} = \frac{T_S}{T_p} = \frac{(1 - F_p) * T_s + F_p * T_s}{(1 - F_p) * T_s + \frac{F_p}{n} * T_s} = \frac{(1 - F_p) + F_p}{(1 - F_p) + \frac{F_p}{n}} = \frac{1}{1 - F_p + \frac{F_p}{n}}$$

Définition 3.26 (Loi d'Amdahl)**.** Soit F_p le pourcentage du temps d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle grâce à n unités de calcul. La **loi d'Amdahl** définie l'accélération théorique en latence par :

$$S_{amdahl} = \frac{1}{1 - F_p + \frac{F_p}{n}}$$

Remarque 3.6. Nous pouvons aussi définir la loi d'Amdahl par rapport au pourcentage F_{np} du temps d'exécution de la tâche qui ne doit pas bénéficier d'une amélioration parallèle (celui séquentiel). Nous avons donc $F_p = 1 - F_{np}$ et :

$$S_{amdahl} = \frac{1}{1 - (1 - F_{np}) + \frac{1 - F_{np}}{n}} = \frac{1}{F_{np} + \frac{1 - F_{np}}{n}}$$

Loi de Gustafson

Nous allons procéder de la même manière qu'avec la loi d'Amdahl mais en utilisant la charge plutôt que le temps d'exécution. Nous avons donc :

$$C_s = (1 - F_p) * C_s + F_p * C_s$$

La loi de Gustafson suppose que la charge de la partie parallèle est améliorée linéairement en fonction du nombre d'unités de calcul n. Le charge total du programme parallèle C_p est donc :

$$C_p = (1 - F_p) * C_s + F_p * n * C_s$$

Pour finir, nous appliquons l'accélération en latence $S_{latence}$ avec un temps d'exécution T fixe :

$$S_{latence} = \frac{T * C_p}{T * C_s} = \frac{C_p}{C_s} = \frac{(1 - F_p) * C_s + F_p * n * C_s}{(1 - F_p) * C_s + F_p * C_s} = \frac{(1 - F_p) + F_p * n}{(1 - F_p) + F_p} = (1 - F_p) + F_p * n$$

Définition 3.27 (Loi de Gustafson)**.** Soit F_p le pourcentage de la charge d'exécution de la tâche qui doit bénéficier d'une amélioration parallèle grâce à n unités de calcul. La **loi de Gustafson** définie l'accélération théorique en latence par :

$$S_{gustafson} = (1 - F_p) + F_p * n$$

Remarque 3.7. Comme pour la loi d'Amdahl, nous pouvons aussi définir celle de Gustafson par rapport au pourcentage F_{np} de la charge d'exécution de la tâche qui ne doit pas bénéficier d'une amélioration parallèle (celui séquentiel). Nous avons donc $F_p = 1 - F_{np}$ et :

$$S_{gustafson} = (1 - (1 - F_{np})) + (1 - F_{np}) * n = F_{np} + n * (1 - F_{np})$$

Moralité

Comme nous pouvons le constater grâce à la figure 3.20, la loi d'Amdahl montre que l'accélération théorique est toujours limitée par la partie de la tâche qui ne peut tirer profit de l'amélioration parallèle. Par conséquent, la limite quand le nombre d'unités de calcul n tend vers l'infini est :

$$\lim_{n \to +\infty} S_{amdahl} = \frac{1}{F_p}.$$

Avec une telle limite, cette loi prédit une accélération monotone à partir d'un certain point et n'est donc pas très optimiste. En revanche, la loi de Gustafson est équivalente à l'accélération linéaire quand la partie parallèle est de 100%. Elle montre qu'augmenter la taille du problème peut être bénéfique avec plus d'unités de calcul. Notons qu'aucune des lois existantes ne prédisent une accélération théorique super-linéaire. Pourtant, en pratique, un petit nombre d'algorithmes parallèles sont super-linéaires. Cela s'explique principalement par la mémoire cache partagée entre les unités de calcul. Celle-ci, se retrouvant en plus petite quantité dans l'exécution séquentielle, est multipliée par le nombre d'unités de calcul dans l'exécution parallèle. De ce fait, un nombre conséquent de données provenant de la mémoire principale dans l'exécution séquentielle, se retrouvent dans le cache, dans l'exécution parallèle. Cela a pour conséquence d'accélérer énormément la vitesse d'exécution parallèle. Dans ce cas, la partie séquentielle d'un algorithme parallèle est aussi améliorée par le cache. Nous pouvons donc imaginer qu'une loi d'Amdahl prenant en compte cette mémoire cache aurait une meilleure limite. Les lecteurs intéressés par les accélérations théoriques peuvent consulter l'article de Sun et Ni (1990). Dans celui-ci, les auteurs ajoutent des contraintes comme les communications et les accès à la mémoire afin d'en extraire des lois plus réalistes.

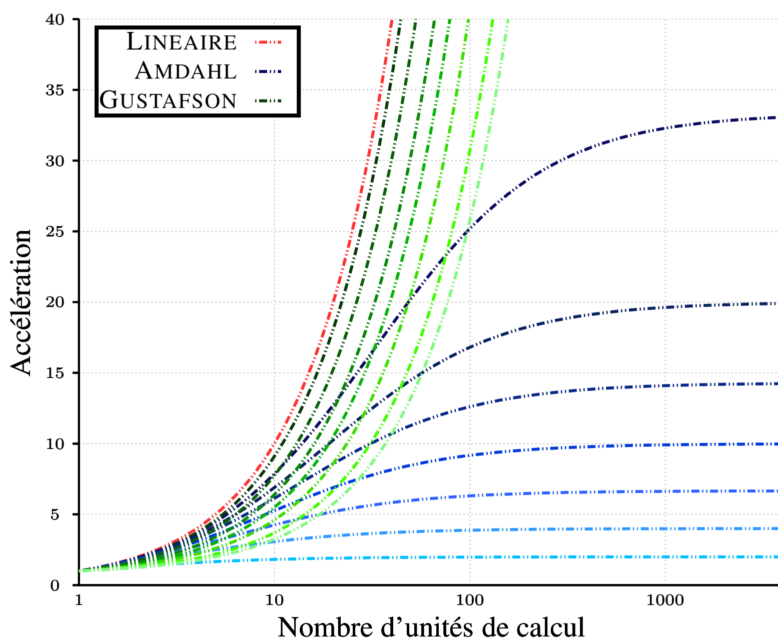


FIGURE 3.20 – Évolution des accélérations théoriques apportées par les lois d'Amdahl et Gustafson par rapport au nombre d'unités de calcul. Pour chacune des lois, les courbes d'une même couleur représentent le pourcentage de la partie parallèle, compris entre 99% et 25%, respectivement de la plus foncée à la plus claire. Échelle logarithmique sur l'axe des abscisses.

3.6 Modèles parallèles

Il y a plusieurs moyens de diviser un problème en de nombreuses tâches. Une classification existe par rapport à la taille de ces tâches. On parle alors de niveaux de parallélisme ou de granularité.

Définition 3.28 (Classification basée sur la granularité)**.** Cette classification se base sur la taille des **grains** : la quantité de calcul impliquée dans un processus en nombre d'instructions. Comme le montre le tableau 3.6, la granularité peut être grossière (entres différents programmes) ou très fine (au niveau des instructions).

Niveau	Granularité	Parallélisme	Division	Communication
Instruction	Fin grain	Le plus haut	En petites tâches	Nombreuses
Boucle	Fin grain	Modéré	Compromis	Compromis
Fonction	Moyen grain	Modéré	Compromis	Compromis
Programme	Gros grain	Bas	En grandes tâches	Peu nombreuses

TABLE 3.6 – Classification basée sur la granularité.

3.6.1 Sortes de parallélismes

Rappelons qu'une tâche représente un travail. Dans le but de construire un algorithme parallèle, le problème doit être divisé en plusieurs tâches. Suivant leurs comportements, nous pouvons extraire deux manières complètement différentes de faire du parallélisme.

Définitions 3.29 (Parallélisme de données et Parallélisme de tâches)**.** Quand les tâches doivent effectuer le même travail mais sur différentes données : cela est nommé le **parallélisme de données**. Au contraire, si les travaux effectués par les tâches sont distincts : cela est appelé le **parallélisme de tâches**.

Il est important de noter que la plupart des programmes réels se situent quelque part entre le parallélisme de tâches et celui de données. Le plus souvent, c'est un parallélisme hybride basé sur un parallélisme de tâches qui est composée de parallélismes de données.

Parallélisme de données

Un grand nombre de problèmes scientifiques implique une grande quantité de données stockées sur un ordinateur. Des instructions équivalentes doivent alors être exécutées sur des données différentes. Même si l'architecture de type SIMD est naturellement associée à ce parallélisme, il n'est pas rare d'utiliser des ordinateurs multi-cœurs de type MIMD afin d'exécuter des instructions différentes sur un grand nombre de données.

Exemple 3.4. Un exemple commun est la multiplication de deux matrices. Soit $A_{i,j}$ et $B_{i,j}$ deux matrices à multiplier et $C_{i,j}$ celle résultante ($i < n$ et $j < n$). Ce problème peut être divisé suivant les lignes i et les colonnes j des deux matrices. De ce fait, chaque unité de calcul va exécuter les mêmes instructions mais sur des données différentes ($C_{1,1} = A_{1,1} + B_{1,1}$ pour la première unité de calcul vectoriel, ..., $C_{n-1,n-1} = A_{n-1,n-1} + B_{n-1,n-1}$ pour la dernière).

Parallélisme de tâche

Quelques problèmes, notamment numériques, doivent souvent être découpés en un certain nombre de tâches ayant différentes fonctionnalités. Ces tâches doivent souvent coopérer et échanger des données. Comme elles ne font pas les mêmes travaux, elles dépendent parfois d'autres tâches.

Exemple 3.5. La triangulation de Delaunay pour un ensemble de points dans un plan est une triangulation telle qu'aucun point n'est à l'intérieur du cercle circonscrit d'un des triangles. Ainsi le problème du raffinement par la triangulation de Delaunay consiste à remplacer certains triangles par d'autres qui

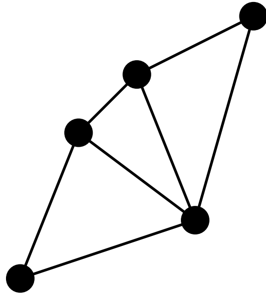


FIGURE 3.21 – Ensemble de triangles non raffinés

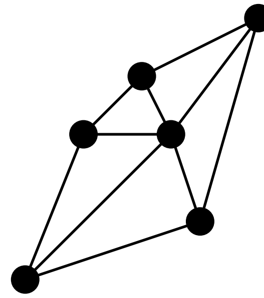


FIGURE 3.22 – Ensemble de triangles raffinés suivant certaines contraintes

respectent la contrainte de Delaunay. Ce problème se généralise en changeant les contraintes attribuées aux triangles (angles, longueurs, positions, ...).

Un simple algorithme parallèle de tâche consiste à définir une tâche comme le raffinement d'un ensemble de triangles. Un maître envoie aux esclaves les potentiels triangles à raffiner sous forme de tâches. Une fois les triangles raffinés, l'information est transmise au maître qui lui donne de nouveaux triangles. Dans le papier de Milind *et al.* (2009), les auteurs exposent plusieurs sortes de parallélisme (de données, de tâches, ...) afin de résoudre ce problème.

3.6.2 Modèles de programmation parallèle

Nous pouvons trouver dans la littérature différentes manières de programmer un algorithme parallèle. Communément nommés modèles de programmation parallèle, ils permettent de classifier les outils de développement logiciel permettant le parallélisme. Même si ces modèles sont indirectement liés aux architectures parallèles, ils ne sont pas spécifiques à un type d'architecture particulier. Par exemple, nous pouvons très bien utiliser le modèle de programmation de la mémoire partagée sur une architecture distribuée. Une telle configuration peut faire apparaître virtuellement à l'utilisateur un seul espace d'adresses englobant la totalité des mémoires principales. Les modèles de programmation font donc une abstraction du matériel utilisé. Avant de décrire ces modèles, nous présentons deux techniques de programmation parallèle employées sur une architecture de type MIMD (section 3.2.6)

Définitions 3.30 (SPMD et MPMD)**.** La technique SPMD (*Single Program Multiple Data*) consiste à exécuter un seul programme (pouvant néanmoins représenter plusieurs flux d'instructions) sur différentes données tandis que celle MPMD (*Multiple Program Multiple Data*) exécute plusieurs programmes distincts.

Remarque 3.8. Ces définitions sont donc associées à la notion de programme où celui-ci possède un seul exécutable. Bien que SPMD utilise un seul programme, celui-ci est souvent décomposé via des branchements conditionnels en plusieurs flux d'instructions afin de permettre un traitement parallèle. La technique SPMD peut donc simuler la MPMD. Notons toutefois qu'il y a une différence jouant sur les performances, dans SPMD, il y a un seul grand fichier binaire exécutable, tandis qu'avec MPMD, il y en a plusieurs petits. De ce fait, la technique MPMD gagne en terme de performance sur une architecture distribuée. Par exemple, un logiciel client/serveur possède souvent deux programmes distincts, un pour le client et un pour le serveur : la technique MPMD a donc été utilisée.

Avec la mémoire partagée

Il existe deux modèles de programmation de la mémoire partagée, un utilisant la notion de processus, l'autre utilisant la notion de *thread*.

Le premier, appelé communication inter-processus (IPC), utilise un grand nombre d'appels systèmes. Ainsi, pour faire un programme parallèle avec ce modèle, il faut créer plusieurs processus, où chacun doit faire une cartographie du même espace d'adresses pour partager des données. Cette dernière manipulation est inefficace car il y a un travail redondant et c'est pourquoi les *threads* ont été créés.

Exemple 3.6. Le standard POSIX apporte une API (bibliothèque de fonctions et procédures (`pthread.h`) pour utiliser la mémoire partagée (`shm_open`, `sem_open`, ...) et UNIX apporte les fonctions gérant la mémoire et les processus (`shmget`, `shmat`, `shmctl`, ...).

Le deuxième définit un *thread*, en français, processus léger ou fil d'exécution qui ressemble du point de vue de l'utilisateur à un processus. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se la partagent. Notamment, les *threads* d'un processus partagent leurs codes exécutables et les valeurs de leurs variables tout le temps. Cela a pour effet de rendre plus efficace certaines manipulations qui ne l'étaient pas sans les *threads* comme le démarrage et l'extinction d'un processus.

Exemple 3.7. Il existe de nombreuses librairies utilisant les *threads* : Intel TBB, Intel Cilk Plus, OpenMP, C++11 Threads, Pthreads, ...

La mémoire partagée peut induire des problèmes liés au partage des ressources.

Définitions 3.31 (Exclusion mutuelle et sections critiques)**.** Lors de l'utilisation de la mémoire partagée, certaines ressources partagées d'un système ne doivent pas être utilisées en même temps (données, fichiers, imprimantes, ...). Ainsi, pour éviter qu'une données soit modifier par plus d'un *thread* en même temps, un mécanisme (algorithme) d'**exclusion mutuelle** doit être mis en œuvre. Ce mécanisme définit des **sections critiques** qui assurent qu'un seul *thread* à la fois les traverses. Ainsi, les données utilisées par plusieurs *threads* à l'intérieure des sections critiques sont protégées des situations concurrentes (*race condition*), situations dans lesquelles le programme serait dans des états imprévus (bogues).

Ils existent plusieurs manières de déclarer des sections critiques, dans nos travaux, nous utilisons la notion de *mutex* (verrou) et de variable condition. Néanmoins, leurs utilisations peuvent entraîner des interblocages (*deadlocks*).

Avec la mémoire distribuée

Le modèle de la mémoire distribuée ou du passage de message doit prendre en compte une architecture distribuée qui utilise un réseau informatique. Il existe deux moyens de communiquer des messages via ce réseau grâce à la notion de *socket* : les *sockets* de flux utilisant le protocole TCP (*Transmission Control Protocol*) et ceux de paquets utilisant le protocole UDP (*User Datagram Protocol*). Les *sockets* de flux possèdent deux voies de communications bi-directionnelles et fiables. TCP s'assure que vos données arrivent séquentiellement et sans erreur, en revanche, UDP ne l'assure pas et un message de confirmation de réception doit être envoyé à l'expéditeur.

Dans le domaine du calcul à haute performance (en anglais, HPC pour *High performance computing*), il existe une spécification officielle nommée MPI pour *Message-Passing Interface* afin de s'abstraire des notions associées aux *sockets* pour ne laisser que celles de messages et de données. MPI est une interface de spécification pour les librairies de passage de messages.

Exemple 3.8. MPI n'est ni un langage, ni une implémentation, c'est une interface guidant un grand nombre d'implémentations : OpenMPI, MPICH, IntelMPI, ... La plupart des ces implémentations utilisent le protocole TCP pour des raisons de fiabilité.

Remarque 3.9. Les librairies qui étendent MPI supportent souvent plusieurs langages. Néanmoins, il n'existe pas d'implémentations MPI qui supportent à la fois le C++ et le Java. Pourtant, des applications Web Client/Serveur pour ces deux langages existent (Architecture REST). Le langage Java ne dispose pas d'une liaison MPI officielle. Toutefois, plusieurs groupes tentent de relier ces deux langages, avec différents degrés de succès et de compatibilités.

Nous pouvons distinguer deux méthodes afin d'échanger des informations dans une telle configuration : la manière « centralisée » et celle « décentralisée ». Il ne faut pas confondre ces notions avec celles d'architectures décentralisées ou centralisées. Contrairement à ces dernières, nous ne parlons pas du matériels utilisés mais bien de la manière de programmer un échange d'informations.

Définition 3.32 (Échange d'information « centralisé »)**.** Dans le modèle de programmation **centralisé** (aussi appelée clients/serveur), lorsqu'une information doit être partagée entre les unités de calcul, elle est d'abord envoyée à une unité de calcul « maître » puis celui-ci renvoie l'information aux autres unités de calcul. Cette méthode permet de centraliser les données avant leur partage afin d'effectuer des calculs sur celles-ci.

Définition 3.33 (Échange d'information « décentralisé »)**.** Dans le modèle de programmation **décentralisé**, lorsqu'une information doit être partagée, l'unité la possédant l'envoi directement aux autres unités de calcul.

Modèles hybrides

Un modèle hybride combine certains des modèles de programmation décrit précédemment. Ces hybridations permettent de tirer profit de plusieurs bibliothèques afin de gagner en performance. Néanmoins, de telles hybridations apportent des problèmes de compatibilité. Par exemple, la notion de *thread-safety* est très importante dans une hybridation et engendre des coûts supplémentaires.

Exemple 3.9. Un exemple commun est la combinaison du modèle de passage de message avec celui des *threads* : OpenMPI/OpenMP, MPICH/pthread, ... Les *threads* réalisent des calculs intensifs en utilisant la mémoire partagée locale des ordinateurs tandis que les communications entre les processus (ou machine) sont faites à travers le réseau en utilisant MPI.

Remarque 3.10. Dans nos contributions présentées dans le chapitre 7, nous faisons une étude de plusieurs modèles de programmation hybride avec différentes manières de faire des communications pour la résolution du problème SAT. Nous montrons l'importance de ce sujet et apportons un solveur plus performant.

Interbloquages

Définition 3.34 (Interbloquage)**.** Un **interblocage** (en anglais, *deadlock*) est un état dans lequel chaque membre d'un groupe est en train d'attendre indéfiniment qu'un ou plusieurs autres membres fassent une action comme la récupération d'un message ou le blocage/déblocage d'un *mutex*.

Ainsi, dans nos travaux, un interblocage peut provenir de deux sources différentes : dans l'utilisation de *mutex* et/ou dans les passages de messages entre plusieurs ordinateurs. Nous avons étudié, dans nos contributions du chapitre 7 plusieurs solutions permettant d'éviter les *deadlocks* par passage de messages dans le cadre de la résolution du problème SAT en parallèle. Le tableau 3.7 expose un tel interblocage via le passage de messages. Dans cet exemple, les deux processus peuvent attendre indéfiniment car l'opération `MPI_SEND` peut être bloquante tant que le message n'est pas reçu par une opération `MPI_RECEIVE`. Une manière d'éviter cet interblocage est alors de sérialiser les communications. Le

Processus 0	**Processus 1**
`MPI_SEND` (a)	`MPI_SEND` (b)
`MPI_RECEIVE` (b)	`MPI_RECEIVE` (a)

TABLE 3.7 – Passage de messages avec interblocage (*deadlock*).

tableau 3.8 expose la sérialisation des messages du tableau 3.7 afin d'éviter le *deadlock*.

Processus 0	**Processus 1**
`MPI_SEND` (a)	`MPI_RECEIVE` (a)
`MPI_RECEIVE` (b)	`MPI_SEND` (b)

TABLE 3.8 – Passage de messages sans interblocage (*deadlock*).

le problème se complique dans les modèles de programmation hybride. En effet, dans cette environnement, plusieurs *threads* peuvent envoyer et/ou recevoir des messages en même temps. Les bibliothèques de la norme MPI doivent alors implémenter des versions dites *thread safety* pour chaque opération de communication (`MPI_SEND`, `MPI_GATHER`, ...). La norme MPI exige qu'une implémentation permettant une telle hybridation doit pouvoir être initialisé via la variable `MPI_THREAD_MULTIPLE`. Quand celle-ci est activée, un processus peut être *multi-threaded* et plusieurs *threads* peuvent faire simultanément plusieurs appels aux opérations de communication sans risque d'interblocage (figure 3.9).

Processus 0		**Processus 1**	
thread 0	*thread* 1	*thread* 0	*thread* 1
`MPI_SEND` (a)	`MPI_RECEIVE` (b)	`MPI_SEND` (b)	`MPI_RECEIVE` (a)

TABLE 3.9 – Passage de messages dans un environnement *multi-threaded*. Une implémentation MPI doit s'assurer que cet exemple d'induit jamais de *deadlocks* quelque soit l'ordre d'exécution des *threads*.

3.7 Conclusion

Nous avons présenté le parallélisme d'une manière générale : les architectures qui le permettent, les concepts qui l'entourent et les modèles de programmation qui le réalisent. L'étude de certains livres a permis la création de ce chapitre. Ainsi, le lecteur intéressé par ces sujets peut consulter les livres de JaJa (1997), Hager et Wellein (2010), Parhami (1999), Eijkhout (2012), Rajasekaran et Reif (2007), El-Rewini et Abd-El-Barr (2005).

Un solveur SAT parallèle possède alors les inconvénients dû au parallélisme (voir section 3.5.2) : un équilibrage de charge, des dépendances entre les tâches et des communications. Ces limitations peuvent alors réduire l'accélération d'un solveur SAT. En effet, nous pouvons remarquer que les clauses apprises dans un solveur CDCL crées des dépendances entre les tâches qui peuvent être exécutées en parallèle. Nos contributions sont très liées aux notions de ce chapitre. En effet, comme tous nos travaux sont massivement parallèle, nous devons faire face à de nombreux problèmes d'encombrement du réseau et de la mémoire partagée. Nous utilisons les notions d'accélération et d'extensibilité d'un programme parallèle afin d'évaluer nos solveurs. Notamment, une de nos contributions (D-SYRUP, chapitre 7) est directement liée aux modèles de programmation distribués et aux problèmes d'inter-blocages présentés dans la section 3.6. Nous allons donc maintenant présenter les solveurs SAT parallèles de l'état de l'art.

Chapitre

4

Résolution parallèle du problème SAT

Sommaire

4.1 Approche « *portfolio* » **90**
4.1.1 MANYSAT : Diversification *Versus* Intensification 91
4.1.2 PLINGELING : De nombreuses techniques *inprocessing* 93
4.1.3 DP^{2}LL : Un solveur déterministe 94
4.1.4 PPFOLIO : Une approche simple 95
4.1.5 BESS : Les bandits manchots 96
4.1.6 PENELOPE : Les clauses gelées 96
4.1.7 MINIRED et GLUCORED : Un raffinement des clauses 97
4.1.8 SYRUP : Un échange paresseux 98
4.1.9 CBPENELOPE et PARACIRMINISAT : Exploiter les communautés . . . 99
4.1.10 TOPOSAT : Les topologies du partage des clauses 100
4.1.11 HORDESAT : Hybridation et diversification 102
4.2 Approches « diviser pour mieux régner » **103**
4.2.1 PSATO : La méthode « chemin de guidage » 104
4.2.2 MTSS : *Thread* riche et *threads* pauvres 106
4.2.3 C-SAT : Plusieurs divisions en concurrence 107
4.2.4 SATCIETY : Du *peer-to-peer* (P2P) 108
4.2.5 PART-TREE-LEARNING : L'échange des clauses 108
4.2.6 CUBEANDCONQUER : Une décomposition statique 109
4.2.7 DOLIUS : Une API simple et clair 112
4.3 Conclusion **113**

DANS ce chapitre, nous présentons les deux principales approches de résolution du problème SAT en parallèle : l'approche « *portfolio* » (section 4.1) et l'approche « diviser pour mieux régner » (section 4.2). Afin de mieux détailler ces approches, nous décrivons de nombreux solveurs. Ces derniers sont présentés dans l'ordre chronologique afin de décrire au fil des ans les améliorations apportées dans chacune des deux approches.

Chaque solveur est présenté par un titre le caractérisant et par une étiquette en dessous du titre précisant les architectures avec lesquelles il est compatible et l'article de base du solveur. Ainsi, l'étiquette « Mémoire partagée » signifie que le solveur est *multi-thread* et ne peut utiliser qu'une seule machine tandis que « Mémoire distribuée » signifie que le solveur peut utiliser plusieurs machines car il implémente un passage de messages entre les machines. Enfin, l'étiquette « Hybride » signifie que le solveur utilise à la fois une bibliothèque *multi-thread* et une de passage de message. Ce dernier est donc, bien sûr, compatible avec plusieurs machines.

Notons que nous comparons, dans nos contributions certains de ces solveurs avec les nôtres. Le lecteur intéressé par le fonctionnement de ces solveurs peut donc dans ce chapitre en avoir des descriptions précises. Ces solveurs sont SYRUP (section 4.1.8), PLINGELING (section 4.1.2), TREENGELING (section 4.2.6) pour ceux *multi-thread* et HORDESAT (section 4.1.11), TOPOSAT (section 4.1.10) et DOLIUS (section 4.2.7) pour ceux compatibles avec plusieurs machines.

4.1 Approche « *portfolio* »

Les approches de type *portfolio* consistent à lancer en concurrence des solveurs CDCL tant qu'aucune solution n'est trouvée. En d'autres termes, elles exécutent plusieurs solveurs en compétition comme dans une course afin d'en déterminer un gagnant : le premier apportant une solution. La figure 4.1 présente cette approche. Dans celle-ci, n solveurs sont lancés pendant un certain temps sur la formule initiale Σ. Dès que le solveur 2 apporte une solution (carré rouge), les autres solveurs s'arrêtent. À ce moment, la recherche des autres solveurs est stoppée (lignes barrées). Chaque solveur consomme son propre temps CPU. Le temps de résolution du problème est alors le temps réel du solveur trouvant la solution (solveur 2). Le premier solveur parallèle *portfolio* est MANYSAT (Hamadi *et al.* 2009c). Cette approche est très efficace sur les instances industrielles puisqu'elle a gagnée de nombreuses compétitions dans cette catégorie. En effet, plusieurs solveurs basée sur ce concept ont obtenu la première place : MANYSAT en 2008 et 2009, CRYPTOMINISAT en 2011, PPFOLIO sur les instances satisfaisables en 2011, PLINGELING en 2013 et 2014, SYRUP en 2015 et 2017.

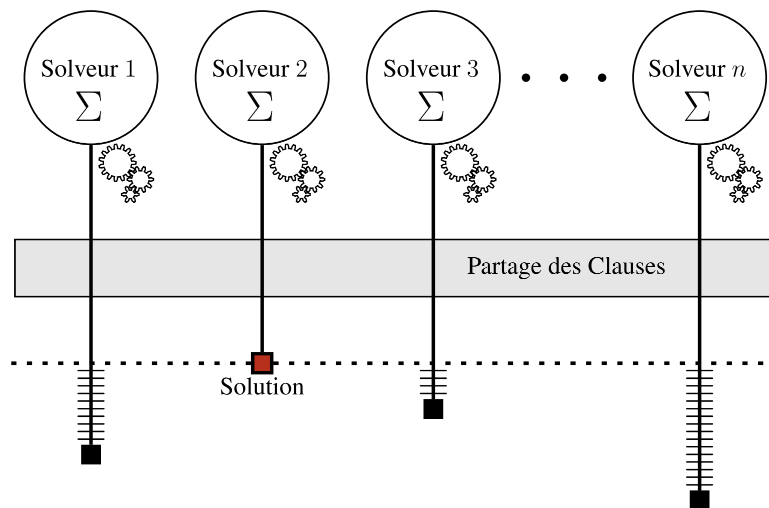


FIGURE 4.1 – Approche « *portfolio* »

Comme nous pouvons le constater, ce qui justifie l'utilisation d'approche de type *portfolio* est que les performances de l'algorithme CDCL sont extrêmement influencées par de nombreux paramètres, notamment ceux utilisés dans les heuristiques de choix de variable et de polarité. En effet, dans cette étape, il existe un grand nombre d'heuristiques possibles et aucune d'entre elles ne dominent toutes les autres sur toutes les instances. Toutes ces configurations peuvent être considérées afin de diversifier l'approche *portfolio*. À titre d'exemple, dans Xu *et al.* (2010) et Aigner *et al.* (2013), une configuration automatique des paramètres des solveurs SAT est mise en place afin d'assurer une bonne diversification de la recherche.

En plus de cette diversification, l'échange des clauses apprises (figure 4.1) entre les solveurs CDCL garanti un gain de performance additionnel. En effet, ces dernières permettent de réduire les recherches redondantes, c'est-à-dire, les travaux effectués en parallèle sur le même espace de recherche. Par conséquent, une clause apprise à partir d'un conflit par un solveur CDCL est ensuite distribuée aux autres solveurs CDCL afin de leurs éviter de reparcourir un même espace de recherche.

Le problème lié aux échanges de clauses est de savoir lesquelles doivent être partagées. En effet, échanger la totalité des clauses est infaisable en pratique, notamment quand le nombre de solveurs séquentiels est élevé. Ces dernières sont trop nombreuses et les échanger toutes ralentirait considérablement

la recherche, plus précisément, cela à un impact important sur la propagation unitaire même en utilisant les structures de données paresseuses !.

Pour pallier ce problème, la solution souvent employée par les solveurs parallèles est d'échanger les clauses suivant une ou plusieurs conditions. Par exemple, le solveur MANYSAT partage uniquement les clauses apprises de taille inférieure à 8. Plus récemment, une technique intéressante appelée « *lazy clause exchange* » est présentée dans la section 4.1.8 (Audemard et Simon 2014).

En plus des problèmes liés à la propagation unitaire, l'échange de clauses peut causer de nombreux ralentissement qui sont intrinsèquement liés au fait qu'il faille communiquer les informations entre les unités de calcul. Afin d'essayer d'endiguer ce problème, l'une de nos contribution a consisté à proposer un modèle de programmation pour l'implémentation de solveur *portfolio* dans un cadre distribué.

4.1.1 MANYSAT : Diversification *Versus* Intensification

▷ Hamadi, Jabbour, et Sais (2009c)
▷ Mémoire partagée

Définitions 4.1 (Diversifier/Intensifier la recherche)**.** Dans l'optique de ne pas effectuer trop de travail redondant, il est souhaitable que les différents solveurs n'explorent pas de la même manière l'espace de recherche. Pour éviter cela, il est nécessaire de choisir des stratégies hétérogènes : c'est la **diversification**. Néanmoins, l'utilisation d'approches totalement orthogonales peut réduire la pertinence des clauses partagées. En effet, lorsqu'un solveur apprend une nouvelle clause, il se trouve dans un certain espace de recherche où celle-ci est utile (propagation du littéral assertif). Lorsque cette clause est partagée avec un autre solveur, il est possible que celui-ci se trouve dans une autre partie de l'espace de recherche, de sorte que cette clause n'a aucun intérêt (satisfaite par exemple). Ainsi, il est parfois utile d'orienter plusieurs solveurs CDCL vers le même espace de recherche afin de profiter du partage des clauses apprises : c'est de l'**intensification**. Le challenge consiste donc à trouver la bonne « distance » entre les différentes unités de calcul, c'est-à-dire, trouver le bon compromis entre intensification et diversification.

Le solveur MANYSAT est le premier solveur parallèle de type « *portfolio* ». Élu meilleur solveur SAT parallèle deux années de suite, respectivement, lors de la SAT-Race 2008 et de la SAT-Compétition 2009, ce solveur diversifie et intensifie la recherche afin d'obtenir des meilleures performances. Les auteurs de MANYSAT changent la configuration des solveurs afin de diversifier la recherche. Dans sa première version, quatre solveurs CDCL été configurés différemment (Hamadi *et al.* 2009c). De plus, les auteurs ont essayé plusieurs manières d'échanger les clauses apprises en se basant sur la taille en nombre de littéraux de celles-ci (≤ 4, ≤ 8, ≤ 12 et ≤ 16). Ils démontrent expérimentalement que partager les clauses de taille ≤ 8 donne les meilleurs résultats. Par ailleurs, des travaux basés sur MANYSAT proposent deux heuristique afin d'ajuster dynamiquement la taille des clauses à partager entre deux unités de calcul ((Hamadi *et al.* 2009a)). Dans cette section, nous présentons une approche proposée dans Guo *et al.* (2010) de type maîtres/esclaves. Dans celle-ci, les maîtres assurent la diversification tandis que les esclaves intensifient la recherche dans certains espaces de recherche fournit par les maîtres.

Diversification

La figure 4.2 expose une telle approche maîtres/esclaves. Dans celle-ci, chaque solveur est configuré différemment, notamment, grâce à trois paramètres connus comme étant très sensibles dans l'algorithme CDCL :

- le choix de variable EVSIDS ;

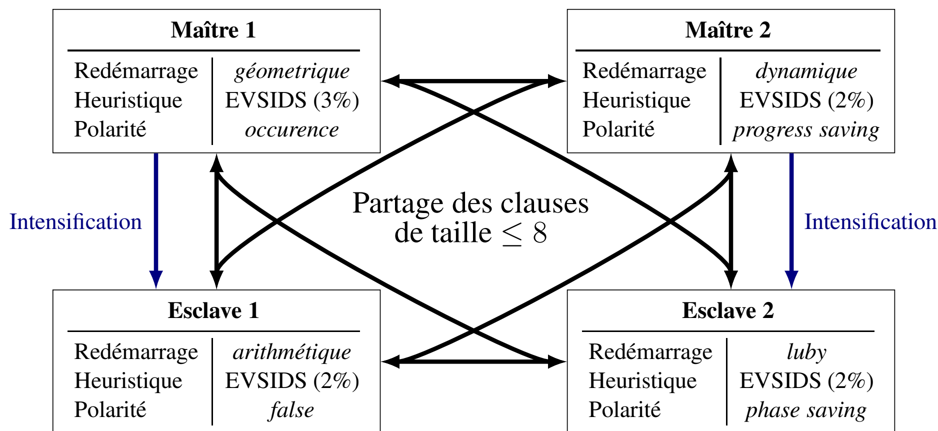


FIGURE 4.2 – Configuration maîtres/esclaves du solveur MANYSAT

- le choix de la polarité ;
- et la politique de redémarrage.

Le choix de variable EVSIDS possède un paramètre très sensible gérant l'adoucissement lié à cette heuristique (c'est le facteur f compris entre 0 et 1 dans le chapitre 2, section 2.3.3, page 43). Pour rappel, cet adoucissement permet de gérer l'oubli des anciens conflits. Ainsi, les poids des variables des anciens conflits se retrouvent, au fur et à mesure des conflits générés, moins importants. Un taux d'oubli de 0% considère que tous les conflits ont la même importance, peu importe les moments de leurs générations durant la recherche. En revanche, plus cette valeur est élevée (par exemple, dans la figure 4.2, 3% contre 2%), moins les anciens conflits sont pris en considération.

L'heuristique de choix de polarité change aussi suivant le cœur de calcul utilisé. Pour cela, trois heuristiques différentes sont étudiées :

- *phase-saving* choisit la dernière polarité obtenue pendant la recherche ;
- *occurence* choisit la polarité contenant le plus d'occurrences dans la base de clauses apprises ;
- *false* choisit toujours le littéral à faux.

Pour finir, les heuristiques de redémarrage enclenchent un redémarrage en fonction du nombre de conflits. Celles-ci permettent de diversifier la recherche et sont définies différemment suivant le cœur de calcul utilisé :

- la suite *géométrique* de raison 1.5 tel que $u_n = 100 \times 1.5^n$;
- une heuristique *dynamique* telle que $x_0 = x_1 = 100$ et $x_i = f(y_{i-1}, y_i)$ quand $i > 1$. Si $y_{i-1} < y_i$, $f(y_{i-1}, y_i) = \frac{1200}{y_i} \times |cos(1 - \frac{y_{i-1}}{y_i})|$, sinon, $f(y_{i-1}, y_i) = \frac{1200}{y_i} \times |cos(1 - \frac{y_i}{y_{i-1}})|$, y_i représente la moyenne de la taille des *backjumps* au redémarrage numéro i. L'idée derrière cette heuristique est alors d'avoir un bon indicateur des décisions faites par erreur durant la recherche ;
- la suite *arithmétique* telle que $x_0 = 16000$ et $x_i = x_{i-1} + 16000$;
- la série de *luby* (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, ...) où chaque terme est multiplié par 512 afin d'effectuer au minimum 512 conflits avant un redémarrage.

Ainsi, nous avons autant de solveurs disponibles que de combinaison de paramètres. Il devient alors facile de diversifier les solveurs en considérant des stratégies différentes.

Intensification

Afin d'intensifier la recherche, un maître transmet à un esclave un ensemble de littéraux apparu dans les analyses de conflits pendant sa recherche (flèche bleue de la figure 4.2). Plus précisément, il s'agit des littéraux apparaissant dans les clauses résolvantes jusqu'à l'obtention du premier UIP. Le but est que l'esclave considère la recherche effectuée par le maître autour des mêmes conflits. Pour cela, quand un esclave récupère cet ensemble de littéraux, il augmente leurs activités EVSIDS afin d'avoir plus de chance de les choisir comme littéraux de décisions. Notons que plusieurs topologies ont été testées afin de savoir quels sont les solveurs maîtres et les solveurs esclaves. Les auteurs ont obtenu les meilleurs résultats en appliquant la topologie représentée par la figure 4.2, c'est-à-dire, celle où un maître guide un seul esclave. Pour finir, notons que l'analyse de conflits dans MANYSAT est étendue. Elle prend en compte les clauses satisfaites dans les graphes d'implications. Cette dernière amélioration n'est pas habituelle et elle apporte une nouvelle manière de diminuer la taille des clauses assertives (Hamadi *et al.* 2009c). Par ailleurs, notons aussi que les travaux de Guo et Lagniez (2011b) considèrent la polarité afin de diversifier la recherche.

4.1.2 PLINGELING : De nombreuses techniques *inprocessing*

▷ Biere (2010)

▷ Mémoire partagée : pThread

PLINGELING est la version *multi-threaded* du solveur séquentiel LINGELING. Afin de diversifier la recherche, chaque *thread* solveur est initialisé différemment. Plusieurs facteurs sont utilisés à cette fin :

- la graine aléatoire ;
- l'ordre des indices des variables ;
- la *phase-saving* ;
- l'effort autorisé par les différentes techniques de prétraitement.

La première version de PLINGELING partage uniquement les clauses unaires (Biere 2010). Pour cela, un *thread* maître contient un *buffer* global de littéraux unaires qui est « paresseusement » synchronisé avec les *threads* solveurs. Quand une clause unaire est déduite par un solveur, celle-ci est ajoutée sans synchronisation via un *buffer* local sauf si ce dernier est plein. Dans ce cas, et aussi quand un *thread* solveur demande les littéraux unaires du *buffer* global, les *buffers* locaux sont transférés dans le *buffer* global entraînant une synchronisation (via un *mutex*). Les *threads* solveurs récupèrent les littéraux du *buffer* global régulièrement.

Définition 4.2 (littéraux équivalents)**.** Si deux clauses c_1 et c_2 appartenant à la formule initiale ou bien déduites pendant la recherche suivent ce motif :

$$c_1 = \{\neg a \vee b\} \text{ et } c_1 = \{\neg b \vee a\}.$$

Alors les littéraux a et b sont dit **équivalents**. En effet, nous avons :

$$c_1 = \{a \rightarrow b\} \text{ et } c_1 = \{b \rightarrow a\} \text{ donc } a \leftrightarrow b.$$

Par conséquent, la variable a peut être remplacée dans toutes les clauses de la formule par la variable b, ou vice-versa.

Plus tard, en plus des clauses unaires, l'échange des littéraux équivalents est ajouté à PLINGELING (Biere 2011). Cela est inspiré des améliorations (Heule *et al.* 2011) apportées dans le solveur de base LINGELING. Dans celui-ci, certaines techniques de prétraitement sont alors aussi désormais réalisées pendant la recherche (*inprocessing*). Cet échange des littéraux équivalents est donc réalisé pendant ces phases *inprocessing*.

Par la suite, Biere (2012) configure ces solveurs avec la même technique de prétraitement pour tous les cœurs de calcul afin de moins diversifier la recherche. À la place, le prétraitement est effectué une seule fois par un *thread* puis cette première instance est clonée afin d'être amenée sur tous les *threads*. Cela réduit l'utilisation de la mémoire sur certaines instances très grandes. Notons aussi que la réplication des *threads* peut être arrêtée lorsque ces derniers utilisent trop de mémoire.

L'année d'après, le partage des clauses apprises est incorporé (Biere 2013). Une clause est partagée si elle contient moins de 40 littéraux et que sa valeur LBD est inférieure ou égale à 8. Contrairement au solveur PENELOPE (section 4.1.6), toutes les clauses exportées sont importées à moins que celles-ci contiennent un littéral éliminé ou bloqué durant les phases *inprocessing* via la *Bounded Variable Elimination* et la *Blocked Clauses Elimination* (voir chapitre 2, sections 2.3.7 et 2.3.7, page 54). Les clauses sont exportées au maître et copiées dans une file globale. Durant la résolution, chaque esclave récupère régulièrement ces clauses, les plus anciennes d'abord. Pendant cette récupération, les clauses « consommées » par tous les solveurs sont supprimées.

Pour finir, notons que le solveur PLINGELING de nos jours n'a pas changé, mis à part l'ajout d'une technique intégrée dans LINGELING, de reconnaissance des instances via la longueur des clauses, le nombre de variables, avant et après le prétraitement.

4.1.3 DP^{2}LL : Un solveur déterministe

▷ Hamadi, Jabbour, Piette, et Sais (2011)
▷ Mémoire partagée

Définition 4.3 (Indéterminisme)**.** Pratiquement la totalité des solveurs parallèles actuels souffrent d'un phénomène appelé l'**indéterminisme**. En effet, la puissance des solveurs SAT parallèles a un prix non négligeable : ne pas pouvoir reproduire les résultats. Plus précisément, quand un solveur parallèle trouve la solution d'un problème, une deuxième résolution de celui-ci peut apporter une autre solution avec un temps de calcul différent. En d'autres mots, relancer un solveur SAT parallèle plusieurs fois sur le même problème engendre de nombreuses solutions pouvant être différentes et des temps de calcul disproportionnés.

L'exclusion mutuelle est une primitive de synchronisation utilisée en programmation informatique pour éviter que des ressources partagées d'un système ne soient utilisées en même temps. L'avantage de cette technique est d'éviter les interblocages (*deadlocks*) sur les ressources dans un système à mémoire partagée. Néanmoins, les appels aux primitives de synchronisation (*mutex*) se comportent différemment suivant l'exécution d'un programme. Étant donné que les échanges des clauses apprises sont effectués lors du déblocage d'un *mutex*, ces partages provoquent cet indéterminisme.

Définition 4.4 (Barrière)**.** Dans un programme parallèle, une **barrière** (synchronisation) est une primitive bloquant un groupe de processus ou de *threads* l'appelant tant que la totalité de ces derniers n'ont pas fait appel à cette primitive.

Hamadi *et al.* (2011) proposent alors de mettre des barrières entre les échanges de clauses apprises. Pour éviter un trop grand nombre de synchronisations des *threads*, ces échanges sont d'abord effectués

tous les x conflits. Les expérimentations montrent alors que fixer la variable x à 100 apportent de bons résultats, mais restent quand même inférieurs au solveur indéterministe. L'intuition donnée par les auteurs est alors que chaque *thread* doit avoir des valeurs x différentes car chaque solveur ne produit pas la même quantité de clauses en un temps donné. Une heuristique dynamique doit donc être mise en oeuvre. Pour cela, les auteurs définissent le nombre de conflits C_i^k déclenchant un échange pour un *thread* i et pour une période k. Au début, $\mathcal{C}_i^0 = 300$, ainsi, le premier échange est déclenché à 300 conflits pour tous les *threads*. Ensuite, le temps que durera les prochaines périodes en nombre de conflits est calculé au moment de l'échange k grâce à :

$$\mathcal{C}_i^{k+1} = 300 + (1 - \frac{\delta_i}{max(\delta)}) \times 300$$

où δ_i est le nombre courant de clauses apprises dans le solveur du *thread* i à l'étape k et $max(\delta)$ est le maximum des δ_i. Ainsi, $\frac{\delta_i}{max(\delta)}$ représente la vitesse d'apprentissage des clauses d'un solveur i par rapport aux autres. Grâce à cette heuristique dynamique, le solveur déterministe DP^{2}LL arrive ainsi à obtenir les mêmes performances que sa version indéterministe.

4.1.4 PPFOLIO : Une approche simple

▷ Roussel (2011)
▷ Mémoire partagée

PPFOLIO est un outil parallèle *portfolio* naïf. Le but de ce solveur est simplement de trouver la limite inférieure (minorant) des performances pouvant être obtenues en utilisant quelques lignes de programmation simple du système. Notons bien qu'aucune communication n'est donc réalisée dans PPFOLIO. Ainsi, le solveur PPFOLIO est équivalent à taper la commande script suivante dans un terminal :

```
solveur 1 &; solveur 2 &; solveur 3 &; solveur 4 &; solveur 5 &;
```

Cinq solveurs ont été choisis sur la base de leurs résultats à la compétition SAT. Dans chaque catégorie (application, conçue (*crafted*) et aléatoire), le meilleur solveur a été sélectionné. De plus, un second solveur a aussi été choisi si il a résolu au moins dix instances non résolues par le premier. La liste des solveurs sélectionnés pour la version PPFOLIO de 2011 est la suivante :

- CRYPTOMINISAT (Soos 2010) ;
- LINGELING ou PLINGELING (Biere 2010) ;
- CLASP (Gebser *et al.* 2007) ;
- TNM (Wei et Li 2009) ;
- MARCH_HI (Heule et van Maaren 2009).

Chaque solveur est placé sur un des cœurs de calcul. Lorsque le nombre de cœurs disponibles est supérieur à 5 (au nombre de solveurs disponibles), le solveur LINGELING est remplacé par sa version parallèle PLINGELING afin qu'il utilise les cœurs restants.

Définition 4.5 (*Virtual Best Solver* (VBS))**.** Souvent utilisé pour comparer des résultats, le ***Virtual Best Solver*** est un solveur imaginaire possédant les meilleurs résultats apportés par un groupe de solveurs réels. Plus précisément, les résultats en temps CPU du VBS d'une instance donnée est le plus petit temps CPU des temps réalisés par les solveurs réels sur cette instance.

Plusieurs observations peuvent être faites grâce aux résultats de PPFOLIO :

- en plus des instances aléatoires, les solveurs de recherche locale sont efficaces sur certaines familles d'instances conçues (*crafted*) ;
- malgré des accès mémoires en concurrence, faire tourner des solveurs différents sur chaque cœur peut être efficace ;
- PPFOLIO est une approximation simple du *virtual best solver* faite grâce au parallèle et donnant de très bons résultats (obtention de plusieurs médailles lors de compétitions).

4.1.5 BESS : Les bandits manchots

▷ Lazaar, Hamadi, Jabbour, et Sebag (2012)
▷ Mémoire partagée

Un bandit manchot (*One-armed bandit*) est l'expression familière d'une machine à sous dans un casino. Le problème de bandits (manchots) classique à k bras (*Multi-Armed Bandit*) originellement proposé par Robbins (1952) représente k machines à sous ayant chacune un bras. Le fait de tirer le bras d'une machine représente alors un choix (aussi appelé une option ou une action).

Définition 4.6 (Problème des bandits manchots à k bras (*Multi-Armed Bandit problem*) (MAB))**.** Un agent choisit à chaque instant une action parmi les K décisions possibles et reçoit une récompense aléatoire tirée selon une distribution déterminée par l'action choisie. Le but du joueur est de maximiser son gain : la somme des récompenses qu'il reçoit au cours d'une séquence des tirages des bras. Dans ce cas, chaque bras est supposé donner des gains qui sont tirés de façon indépendantes à partir d'une répartition fixe et inconnue. Comme les distributions de récompenses diffèrent de bras à bras, l'objectif est de trouver le bras avec le meilleur gain le plus tôt possible, et de ne se focaliser que sur ce dernier.

Les auteurs se sont appuyé sur ce concept afin d'améliorer les performances du solveur *portfolio* MANYSAT. Comme dans celui-ci tous les cœurs de calcul échangent les clauses apprises dont la taille ne dépasse pas une certaine limite, les coûts de communication de cette approche ont tendance à croître proportionnellement avec le nombre de cœurs. Afin de résoudre ce problème, les auteurs de Lazaar *et al.* (2012) proposent d'utiliser un MAB qui sélectionne les cœurs autorisés à partager leurs informations afin de décongestionner le canal de communication. Pour cela, les auteurs ont implémenté un solveur parallèle nommé *Bandit Ensemble for parallel SAT Solving* (BESS) qui s'inspire du problème du bandit manchot.

L'approche BESS est basée sur les notions émetteur et récepteur. Un émetteur envoie les clauses apprises tandis qu'un récepteur les reçoit. Un émetteur est dit vivant dans une certaine période s'il est autorisé à envoyer des clauses, sinon, il est dit dormant. Au commencement de chaque période, un MAB particulier décide si un émetteur vivant reste en vie, dans le cas contraire, il devient un émetteur dormant et le plus vieux émetteur dormant est réveillé. Voici comment chaque période est définie :

- Le gain instantané d'un émetteur en vie par rapport à un récepteur est défini en fonction d'une heuristique semblable à EVSIDS. Grâce à celle-ci, les auteurs mettent à jour le gain cumulé de chaque émetteur en vie ;
- Le solveur met à jour un seuil de vitalité ;
- L'émetteur vivant devient en sommeil si son gain cumulé est inférieur au seuil de vitalité et, dans ce cas, l'émetteur en sommeil qui a été endormi le plus longtemps est réveillé.

4.1.6 PENELOPE : Les clauses gelées

▷ Audemard, Hoessen, Jabbour, Lagniez, et Piette (2012)
▷ Mémoire partagée : openMP

Le solveur PENELOPE (pour *Parallel Lbd Psm solver*) est basé sur l'heuristique PSM (Audemard *et al.* 2011) et l'heuristique LBD. Pour rappel, ces dernières sont chacune associées à une politique de suppression des clauses apprises (voir chapitre 2, section 2.3.5 et 2.3.5). Habituellement, le critère d'échange des clauses apprises entre les solveurs est seulement basé sur les informations possédées par leurs envoyeurs, c'est-à-dire, que les *threads* receveurs acceptent n'importe quelles clauses de la part des *threads* envoyeurs. À titre d'exemple, dans le solveur MANYSAT, une clause apprise est envoyée aux autres unités de calcul uniquement si sa taille est inférieure ou égale à 8. L'idée apportée par les auteurs de PENELOPE est alors d'étendre ces critères (certaines heuristiques d'échanges de clauses apprises) aux receveurs.

Notons d'abord que la stratégie de redémarrage est celle employée dans GLUCOSE (voir, chapitre 2, section 2.3.6, Audemard et Simon (2009a)). D'un côté (envoyeur), une clause est exportée aux autres *threads* dès que sa valeur LBD est ≤ 8. Notons que la valeur LBD d'une clause peut varier pendant la recherche. De l'autre côté (receveur), une clause fraîchement reçue est directement évaluée par l'heuristique PSM comme si celle-ci était générée localement par le solveur. Pendant cette évaluation, si elle est considérée comme pertinente via l'heuristique PSM, elle est ajoutée dans la base des clauses apprises, sinon elle est gelée. Pour rappel, une clause est pertinente via l'heuristique PSM si sa valeur PSM est petite (≤ 2). Ainsi, l'espace de recherche du solveur recevant la clause est pris en compte. Grâce à cette technique, le solveur PENELOPE a été placé dans le top 3 des compétitions SAT 2012, 2013 et 2014.

4.1.7 MINIRED et GLUCORED : Le raffinement des clauses

▷ Wieringa et Heljanko (2013)
▷ Mémoire partagée

L'idée originale apportée par les auteurs Wieringa et Heljanko (2013) est d'utiliser un *thread* afin de simplifier les clauses apprises. Pour cela, ils proposent une architecture composée de seulement deux *threads* appelés le solveur et le simplificateur. Le solveur est un solveur SAT séquentiel quelconque qui doit implémenter plusieurs méthodes afin d'interagir avec le simplificateur de clauses. Comme le montre la figure 4.3, l'interaction entre ces deux *threads* est limitée à passer des clauses à travers deux structures de données de la mémoire partagée. Plus précisément, ces deux structures sont des files, l'une apporte des clauses apprises par le solveur au simplificateur et l'autre est utilisée afin de transmettre des clauses simplifiées dans le sens opposé.

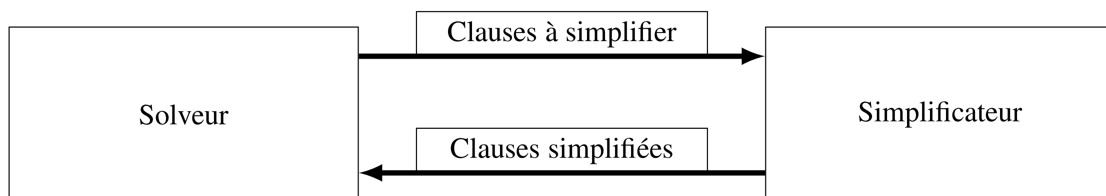


FIGURE 4.3 – Architecture des solveurs MINIRED et GLUCORED

Le solveur

Toutes les clauses apprises par le solveur sont, par défaut, introduites dans la file afin d'être transmises au simplificateur. Néanmoins, cette file est limitée à une taille de 1000 clauses. De ce fait, quand elle est pleine, le solveur remplace la plus ancienne clause dans la file par la nouvelle. Ceci permet de traiter en priorité les clauses les plus récentes.

De plus, les auteurs précisent que la qualité d'une clause change durant la recherche. Plus précisément, une clause étant relativement longue (« mauvaise ») à un moment donnée de la recherche peut devenir relativement courte (« bonne ») un peu plus tard à cause de l'affectation de certains littéraux. Inversement, celle-ci peut être de moins en moins utile, si, par exemple, elle est de plus en plus satisfaite dans l'espace de recherche la concernant. Ce phénomène est connu dans la littérature, par ailleurs, il est aussi observé via l'heuristique LBD. En effet, la valeur LBD d'une clause peut être améliorée pendant la recherche. Pour traiter ce problème, les auteurs proposent de trier la file grâce à une heuristique mesurant la qualité des clauses. C'est pourquoi ce papier donne lieu à deux solveurs : l'un utilise une heuristique basée sur de la taille des clauses (MINIRED) et l'autre utilise l'heuristique LBD (GLUCORED). Ce choix est facilité par le fait que les solveurs séquentiels utilisés de base (MINISAT et GLUCOSE) implémentent respectivement et d'une manière disjointe, ces deux heuristiques. Par conséquent, dans MINIRED (resp. GLUCORED), lorsque le simplificateur récupère une clause, c'est celle ayant la plus petite taille (resp. la plus faible valeur LBD) en priorité.

Le simplificateur

L'algorithme du simplificateur est basé sur la propagation et l'apprentissage des clauses. Simplement, celui-ci assigne les littéraux d'une clause prise en entrée à `faux` un par un jusqu'à ce que toutes les variables soient assignées (voir la technique de prétraitement appelée vivification, section 2.3.7, Piette *et al.* (2008)). Pendant cette démarche, des conflits peuvent survenir et des *backjumps* sont alors réalisés.

Premièrement, lorsqu'un conflit apparaît, une analyse de ce conflit à lieu et une nouvelle clause est apprise. Cette clause est ajoutée à une base de clauses apprises par le simplificateur. Remarquons que plusieurs conflits peuvent avoir lieu pendant l'affectation de tous les littéraux de la clause en entrée. Par conséquent, plusieurs clauses peuvent être apprises lors de cette démarche. Remarque importante, les auteurs ne transmettent pas ces nouvelles clauses au solveur, mais elles sont cruciales afin d'améliorer les performances du simplificateur.

Deuxièmement, quand toutes les variables de la clause à traiter sont assignées, celle-ci peut, par moment, être réduite. La clause réduite en sortie est obtenue en supprimant les littéraux de la clause en entrée impliquées par le solveur plutôt qu'assignés. Plus précisément, un littéral est impliqué par le solveur lorsque celui-ci est propagé par la propagation unitaire via une autre clause α. Cela est une implémentation du prétraitement *self-subsumption resolution* (voir section 2.3.7, page 54). C'est un raisonnement rigoureux (ou correct) car la résolution entre la clause en entrée et une autre α entraîne une clause résolvante impliquée par la formule initiale.

Pour finir, notons que les auteurs ont montré expérimentalement que GLUCORED est beaucoup plus efficace que MINIRED. De plus, les auteurs montrent aussi que GLUCORED composé de seulement deux THREADS est meilleur en nombre d'instances résolues que le solveur PENELOPE avec 4 *threads* sur les instance insatisfaisables. Notons qu'une approche similaire minimisant les clauses apprises grâce à une technique *inprocessing* dans un solveur séquentiel (section 2.3.8, page 57) a gagnée la compétition SAT 2017 (*main track*).

4.1.8 SYRUP : Un échange paresseux

▷ Audemard et Simon (2014)

▷ Mémoire partagée : pThread

Le solveur SYRUP est le solveur parallèle *portfolio* basé sur GLUCOSE. Dans ce solveur, les clauses sont échangées en utilisant la mémoire partagée. La figure 4.4 donne une vue générale de la gestion du partage des clauses entre n solveurs séquentiels GLUCOSE. Voici comment les clauses sont importée et exportées. Quand un solveur décide de partager une clause (les flèches rouges en pointillés), elle est ajoutée dans un *buffer* localisé dans la mémoire partagée. SYRUP tente de réduire le nombre de clauses partagées entre les cœurs de calcul en envoyant uniquement les clauses qui sont potentiellement utiles. Les clauses exportées sont alors les clauses unaires, binaires et collantes (*glue*) mais aussi celles étant vues au moins deux fois durant les analyses de conflits. Il est important de noter que le *buffer* de la mémoire partagée est limité (par défaut, à $0.4MB\times$ le nombre de *threads*) et qu'une clause est supprimée du *buffer* quand tous les *threads* l'ont importée (récupérée). Par conséquent, si le *buffer* est plein au moment de l'exportation d'une clause, cette dernière n'est simplement pas échangée. Cela permet de ne pas surcharger la mémoire partagée et ainsi ne pas ralentir les solveurs quand le nombre de clauses à partager est trop élevé.

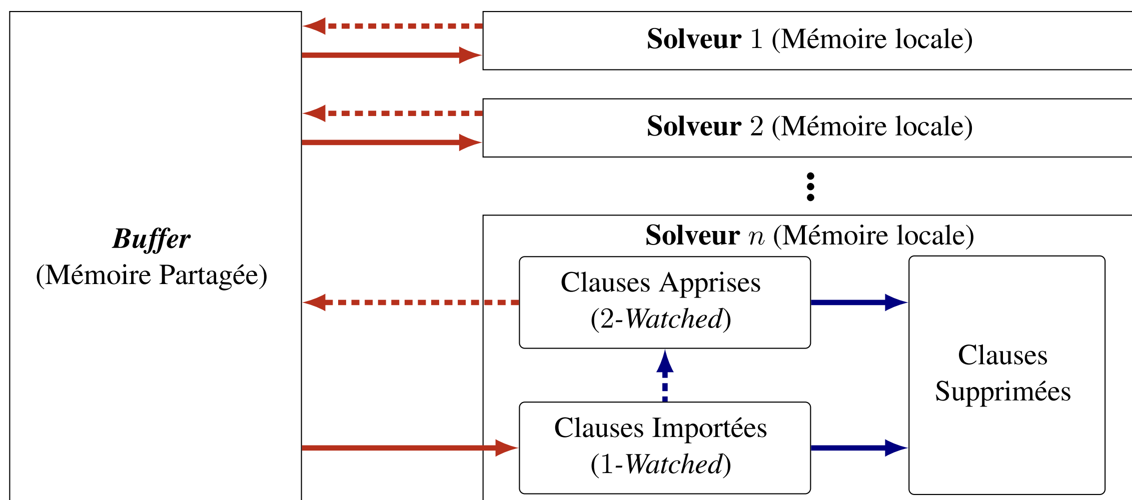


FIGURE 4.4 – Architecture du solveur *portfolio* SYRUP

L'importation est réalisée à chaque redémarrage d'un solveur. Comme pour le solveur PENELOPE, les clauses importées (représentées par les flèches rouges pleines) ne sont pas directement ajoutées dans la base de clauses apprises d'un solveur. Un traitement spécial leurs est réservées via une structure de données paresseuse ressemblant à celle des solveurs CDCL dite 2-*Watched* (voir chapitre 2, section 2.3.4, page 45). Cette structure permet d'avoir pour chaque clause importée, une variable dite sentinelle (*watch*). Ce mécanisme, alors appelé 1-*Watched*, est suffisant pour assurer la détection de toutes clauses conflictuelles pendant la propagation unitaire. Néanmoins, il n'est pas possible de détecter les clauses devenant unaires et devant donc être propagées par la propagation unitaire. Toutefois, en détectant uniquement les conflits, la structure 1-*Watched* a l'avantage d'être beaucoup plus rapide en temps de calcul. Elle est donc un très bon candidat pour une heuristique permettant de juger la qualité d'une clause fraîchement importée. Ainsi, quand une clause importée est en conflit, elle est traitée comme une clause apprise localement et est intégrée dans la base des clauses apprises du solveur. Elle devient donc 2-*Watched* et peut donc, désormais, propager des littéraux pendant la recherche (cela est représenté par les flèches bleues en pointillé). Cela permet de sélectionner dynamiquement les clauses utiles lors de

leur réception, c'est-à-dire, celles pertinentes en fonction de l'espace de recherche courant d'un solveur receveur (comme le solveur PENELOPE avec PSM). Comme le nombre de clauses importées peut être, par moment, très élevé, SYRUP utilise une stratégie dédiée qui supprime périodiquement les clauses qui sont hypothétiquement moins utiles dans la recherche (flèches bleues pleines).

4.1.9 CBPENELOPE et PARACIRMINISAT : Exploiter les communautés

▷ Sonobe, Kondoh, et Inaba (2014)
▷ Mémoire partagée

Le but des auteurs de Sonobe *et al.* (2014) est de différencier les variables cibles (de décisions) de chaque solveur séquentiel via les communautés afin de diversifier la recherche. Ces solveurs forment alors un solveur parallèle *portfolio*. Pour attribuer des variables cibles à un solveur, l'activité de chacune d'entre elles est alors simplement augmentée via l'heuristique EVSIDS. Remarquons que, même si cette manière de résoudre est considérée comme une approche de type *portfolio*, elle est très liée à l'approche « diviser pour mieux régner ». Elle est pourtant *portfolio* dans le sens où il n'y a pas de division précise sous forme d'un arbre de division (voir section 4.2).

Définition 4.7 (Communautés)**.** D'une manière abstraite, les instances industrielles sont dites très structurées. En effet, les clauses et les variables d'une instance CNF sont liées par leurs natures. Ces liaisons sont plus denses pour certaines clauses ou variables que d'autres. Ces variations peuvent être exprimées par un ensemble de communautés. Une **communauté** est alors un ensemble de variables fortement lié par la structure de l'instance, plus précisément, par les clauses. Il existe de nombreuse méthodes permettant de calculer les communautés.

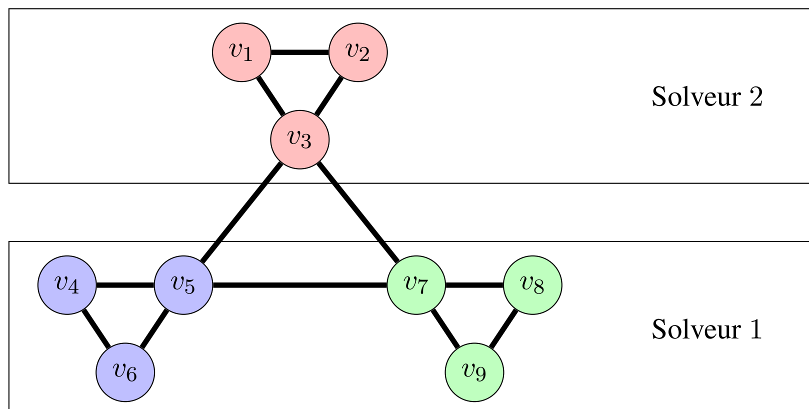


FIGURE 4.5 – Graphe d'incidence des variables et trois communautés. Dans CBPENELOPE et PARACIRMINISAT, chaque communauté est attribuée à un solveur. Dans cette figure, deux unités de calcul sont utilisées représentant alors deux solveurs séquentiels.

Dans Sonobe *et al.* (2014), les communautés sont calculées via le *Variable Incidence Graph* (VIG). Dans celui-ci, chaque nœud est une variable et chaque arrête est représentée par l'existence d'une clause reliant deux variables. La figure 4.5 représente le VIG d'une instance composée de 9 variables. Nous pouvons alors en déduire trois communautés (représentées par trois couleurs différentes dans cette figure). Ces communautés sont calculées à l'aide de l'algorithme GFA (pour *Graph Folding Algorithm*) étudié dans Blondel *et al.* (2008). Dans les deux solveurs CBPENELOPE et PARACIRMINISAT, ces communautés sont recalculées régulièrement en prenant en compte les clauses apprises en plus des clauses initiales au fur et à mesure de la recherche. Chaque communauté est assignée à un solveur (comme dans

la figure 4.5). Quand il y a plus de communautés que de solveurs, plusieurs communautés sont affectées à un même solveur. En revanche, dans le cas contraire, quand il y a moins de communautés que de solveurs, ceux sans communautés résolvent l'instance normalement. Ces mécanismes sont implémentés sur la base de deux solveurs parallèles PENELOPE et PARAMINISAT. Les deux solveurs basés sur les communautés sont alors respectivement appelés CBPENELOPE et PARACIRMINISAT.

4.1.10 TOPOSAT : Les topologies du partage des clauses

▷ Ehlers, Nowotka, et Sieweck (2014)
▷ Mémoire distribuée : MPI

Les topologies

Les auteurs Ehlers *et al.* (2014) étudient différentes topologies permettant l'échange des clauses dans un environnement massivement parallèle. Pour cela, ils définissent un graphe de communication où chaque solveur correspond à un nœud. Les arêtes représentent des contraintes de communication. Plus précisément, une communication (un partage de clauses) est autorisée uniquement si une arête relie deux nœuds (deux solveurs). Le graphe ainsi défini représente alors une certaine topologie de communication. Dans ce solveur parallèle nommé TOPOSAT, ces topologies permettent de contrôler le nombre total de communication effectué entre plusieurs instances du solveur séquentiel bien connu GLUCOSE.

TOPOSAT utilise la librairie de communication OpenMPI afin d'effectuer les communications. Néanmoins, l'utilisation des communications collectives (communications partageant des données entre toutes les machines) n'est plus possible à cause de l'utilisation des topologies. Cela est un léger inconvénient car ces communications collectives sont considérées comme étant plus efficaces quand elles sont utilisées par plusieurs processus. Les auteurs utilisent alors des communications dites « point à point » (processus à processus). Dans TOPOSAT, chaque processus est composé d'un *thread* pour un solveur glucose et d'un autre pour faire les communications sur un même cœur de calcul.

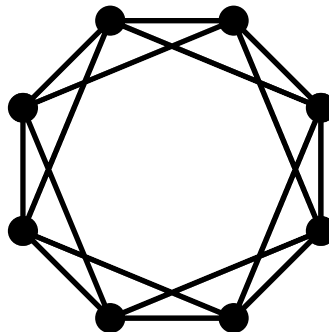


FIGURE 4.6 – Topologie avec 8 processus autorisant un solveur à communiquer avec 4 voisins dans TOPOSAT.

Les topologies étudiées par les auteurs sont définies en fonction d'un nombre de voisins. L'idée est que les échanges sont alors autorisés entre un nœuds et un certain nombre de nœuds voisins. La figure 4.6 montre une telle topologie avec 8 processus (8 *threads* communicateurs et 8 *thread* solveurs de type GLUCOSE) et un nombre de voisins égal à 4. Dans cette figure, les échanges autorisés entre les solveurs sont alors représentés par les arêtes entre les nœuds. Les auteurs montrent alors expérimentalement qu'un nombre de voisins égal à 32 donne les meilleurs résultats quand le nombre d'unités de calcul est de 256 cœurs. Inversement, ils montrent que le solveur obtient une efficacité parallèle décroissante sur 128 et 256 cœurs si la topologie mise en oœvre est celle d'autoriser tous les solveurs à échanger toutes les clauses entre eux (cela est alors une topologie sans aucune contrainte).

Les communications

Remarquons que deux librairies de communication sont utilisées, l'une gérant de la mémoire partagée (pThread), l'autre de la mémoire distribuée (OpenMPI). Néanmoins, ce solveur ne peut par être considéré comme hybride. Cela est dû au fait que la librairie *multi-thread* n'est pas utilisée dans l'échange des clauses entre *thread*. En effet, cette librairie sert uniquement à pouvoir faire les communications en même temps que les résolutions. Autrement dit, aucune clause n'est échangée entre les *threads* solveurs via la mémoire partagée, même quand deux *threads* solveurs sont sur la même machine. Celles-ci sont uniquement partagées par le passage de message de la librairie MPI. Une de nos contributions est alors d'examiner différents modèles de programmation possibles (chapitre 7). Plus précisément, nous démontrons expérimentalement qu'il est possible d'être plus efficace dans la manière de faire ces communications.

Afin de diversifier l'espace de recherche, notons que chaque instance de GLUCOSE est initialisée avec une graine aléatoire différente basée sur l'identifiant du processus MPI. Les *threads* communicateurs s'occupent de plusieurs points :

- Envoyer les clauses unaires et binaires à tous les autres processus ;
- Envoyer les clauses possédant une valeur LBD inférieure ou égale à 4 en fonction d'une topologie donnée préalablement ;
- Récupérer les clauses en les filtrant via un hachage permettant de supprimer certaines clauses subsumées (redondantes).

Une boucle effectuant ces tâches est répétée tant que la solution n'est pas trouvée. Cela est ce que nous appelons, dans nos contributions (chapitre 7), un cycle de communication :

- Exécution de toutes les communications des données d'un *buffer* (`MPI_SEND` et `MPI_RECEIVE`) ;
- Attente des terminaisons des communications (`MPI_WAIT_ALL`) ;
- Attente de 5 secondes. Durant ce temps, le *thread* solveur remplit le *buffer* de clauses.

4.1.11 HORDESAT : Hybridation et diversification

▷ Balyo, Sanders, et Sinz (2015)
▷ hybride : pThread+MPI

HORDESAT est l'un des premiers solveurs SAT parallèles *portfolios* dit hybride. Cela signifie que deux librairies de communication coopèrent afin d'échanger les clauses apprises entre les différentes unités de calcul. L'une gérant les communications entre plusieurs cœurs de calcul sur une même machine via la mémoire partagée et l'autre gérant les passages de messages entre différentes machines via le réseau. Notons que nos contributions incluent une étude des modèles de programmation, dont le modèle hybride, dans la résolution du problème SAT. Des précisions sur l'hybridation sont alors apportées dans le chapitre 7.

L'échange des clauses dans HORDESAT sur le réseau est réalisé périodiquement grâce à des cycles de communication. En revanche, contrairement à TOPOSAT, une même quantité de clauses est échangée à chaque cycle (1500 entiers dans leur implémentation). Notons que cette limite est la même sur toutes les machines. Cela est réalisé via une communication collective impliquant la totalité des machines via la routine `MPI_ALLGATHER`.

Chaque processus prépare le message à envoyer aux autres machines en collectant via la mémoire partagée les clauses apprises des solveurs (cœurs de calcul) dans un *buffer* de taille fixe (1500 entiers). Les clauses de taille courte sont mises en priorité dans ce *buffer*. Cela permet de ne pas écarter des

bonnes clauses. En effet, quand ce *buffer* est plein, les clauses restantes ne sont pas partagées. À l'inverse, quand le *buffer* n'est pas plein au moment de la communication, des zéros sont injectés dans ce *buffer* afin qu'il atteigne sa taille fixe de 1500 entiers. Pour remédier à ce problème, quand cette situation arrive, une fonction spéciale (*increaseClauseProduction*) est alors appelée afin d'augmenter d'une manière heuristique la quantité de clauses à partager. Ainsi, quand le *buffer* n'est pas plein, au prochain cycle de communication, il a plus de chance de l'être.

Notons aussi que les clauses insérées dans le *buffer* sont filtrées afin d'éliminer celles redondantes. Pour cela, les auteurs utilisent une fonction de hachage permettant de repérer celles subsumées ou égales en terme de littéraux :

$$\mathcal{H}_i(c) = \bigoplus_{\ell \in c} \ell \times premiers[abs(\ell \times i)\ mod\ |premiers|]$$

où i est une variable choisi librement, $\oplus$ représente l'opération binaire « ou exclusif », $premier[\,]$ est un tableau de nombre premiers et $|premiers|$ est la taille de ce tableau. Cette fonction de hachage assure que deux clauses c_1 et c_2 ne sont pas subsumées ou égales quand $\mathcal{H}_i(c_1) \neq \mathcal{H}_i(c_2)$.

Pour finir, les auteurs de Balyo *et al.* (2015) diversifient la recherche de chaque solveur séquentiel. Pour cela, la phase par défaut de chaque variable est mise aléatoirement sur seulement un des solveurs. Par exemple, si la variable a est fixée aléatoirement dans le premier solveur, elle ne sera pas mise aléatoirement dans les autres. De plus, ils utilisent aussi une diversification intégrée dans PLINGELING basée sur 16 paramètres différents.

4.2 Approches « diviser pour mieux régner »

Les approches « diviser pour mieux régner » (aussi appelées collaboratrices) consistent à diviser récursivement l'arbre de recherche en plusieurs sous-problèmes (figure 4.7). Plusieurs approches sont proposées dans la littérature. Une des plus utilisées, est intitulée le chemin de guidage. Elle découpe l'arbre de recherche sous forme de chemin afin de guider des solveurs vers les sous-problèmes. Cette méthode apporte néanmoins un désavantage : les charges de calcul engendrées par les sous-problèmes sont déséquilibrées. En effet, la division d'un arbre de recherche en deux parties peut induire deux sous-problèmes totalement différents en terme de temps de résolution : l'un très facile se résolvant en quelques secondes et l'autre beaucoup plus difficile pouvant rester indéterminé. La décomposition en sous-problèmes peut se faire de deux manières différentes.

La première dite « **statique** » décompose le problème en sous-problèmes avant la résolution. Elle est employée dans le solveur CUBEANDCONQUER. Ce dernier, associé à quelques techniques de prétraitement (notamment l'élimination des clauses bloquées) réalisées sur le problème initial et les sous-problèmes, a été utilisé afin de résoudre le problème des triplets pythagoriciens booléens (section 1.4.1, page 20, Heule *et al.* (2016)). Dans cette méthode, un très grand nombre de sous-problèmes est généré afin de réduire leurs difficultés et ainsi les résoudre plus rapidement en parallèle. Cependant, cette division en sous-problèmes peut prendre beaucoup de temps même si celle-ci est limité à un certain nombre de sous-problèmes. Et ce temps de division peut, sur certains problèmes, être plus long que leur résolution.

La deuxième est dite « **dynamique** » car elle découpe le problème en sous-problèmes pendant leur résolution. Quand un solveur est inoccupé et qu'il ne reste plus de sous-problèmes à résoudre, une politique de rééquilibrage des charges doit être mise en place. La plus connue est le *work stealing* : quand un solveur est inoccupé, il « vole » une partie d'un sous-problème à un autre solveur en le divisant en

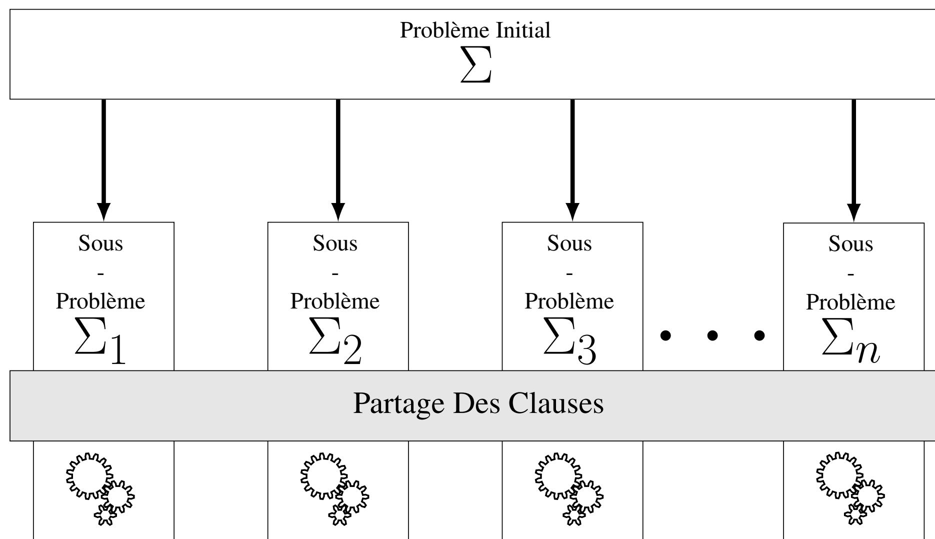


FIGURE 4.7 – Approche « diviser pour mieux régner »

deux. Néanmoins, quand tous les sous-problèmes crées dynamiquement deviennent trop facile, le solveur parallèle effectue alors un très grand nombre de « vols de travail » ralentissant ainsi la recherche : cela est appelée l'effet *ping-pong*.

Contrairement à la méthode concurrentielle, ces deux méthodes « diviser pour mieux régner » doivent attendre que tous les solveurs terminent leurs recherches pour prouver qu'une instance est insatisfaisable. En effet, il faut prouver que la totalité des sous-problèmes sont insatisfaisables afin de démontrer que l'instance initiale est insatisfaisable. En revanche, pour démontrer qu'une instance est satisfaisable, il suffit de montrer qu'un des sous-problème est satisfaisable.

4.2.1 PSATO : La méthode « chemin de guidage »

▷ Zhang, Bonacina, et Hsiang (1996)

▷ Mémoire distribuée

Dans cette section, nous présentons une des méthodes les plus populaires des approches « diviser pour mieux régner ». Intitulée « chemin de guidage » (*Guiding Path*), elle a été proposée par Zhang *et al.* (1996) et par Zhang et Bonacina (1994) via le solveur PSATO. Cette approche consiste à conserver des chemins de recherche permettant d'indiquer quels sont les sous-arbres à développer afin de diviser l'instance. Un tel chemin est représenté par un ensemble de i couples (ℓ_i,α_i), ou ℓ_i est un littéral à propager et α_i est une variable booléenne indiquant quelles sont les branches en cours de résolution parmi les deux sous-arbres (sous-espace de recherche) disponibles à partir de la variable provenant du littéral ℓ_i. La variable booléenne α_i est définie par :

- $\alpha_i = \texttt{faux}$ $\Leftrightarrow$ les deux sous-arbres sont en cours de traitement ou traités ;
- $\alpha_i = \texttt{vrai}$ $\Leftrightarrow$ un seul des deux sous-arbres est en cours de traitement ou traité.

Exemple 4.1. Le chemin de guidage du solveur 1 de la figure 4.8 est $\{(x_1,\texttt{vrai}),(x_2,\texttt{vrai}),(\neg x_3,\texttt{faux})\}$. À partir de celui-ci, il est possible d'obtenir un nouveau chemin de guidage afin d'accueillir le solveur 2. Pour cela, le solveur 1 modifie sont chemin en $\{(x_1,\texttt{faux}),(x_2,\texttt{vrai}),(\neg x_3,\texttt{faux})\}$ et le solveur 2 s'associe au chemin $\{(\neg x_1,\texttt{faux})\}$. Par la suite, si un troisième solveur est inutilisé, nous pouvons appliquer le même raisonnement sur la variable x_2 afin d'avoir un troisième chemin de guidage :

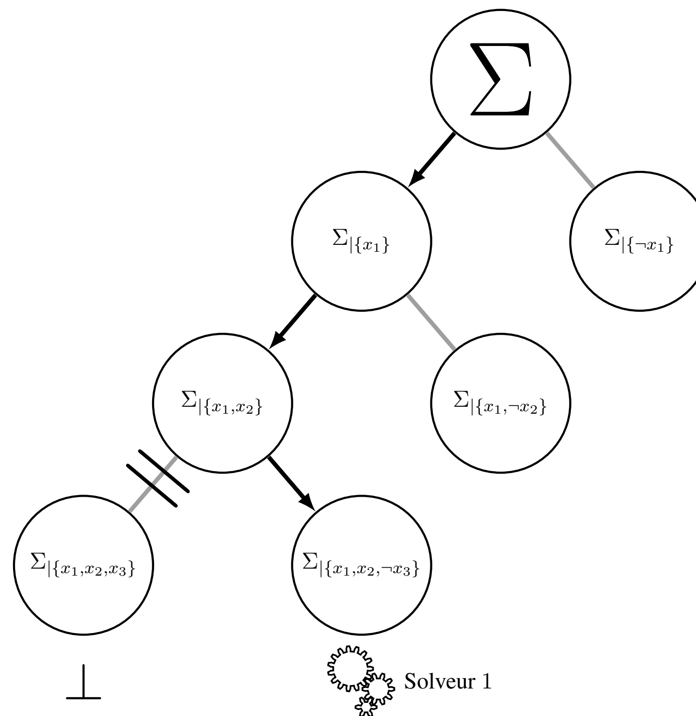


FIGURE 4.8 – La méthode « chemin de guidage ». À ce moment, seul le Solveur 1 travail. Il résout le sous-problème associé au chemin de guidage $\{\,(x_1, \texttt{vrai}\,), (x_2, \texttt{vrai}\,), (\neg x_3, \texttt{faux}\,)\,\}$.

- le solveur 1 a le chemin $\{\,(x_1, \texttt{faux}\,), (x_2, \texttt{faux}\,), (\neg x_3, \texttt{faux}\,)\,\}$;
- le solveur 2 a le chemin $\{\,(\neg x_1, \texttt{faux}\,)\,\}$;
- le solveur 3 a le chemin $\{\,(x_1, \texttt{faux}\,), (\neg x_2, \texttt{faux}\,)\,\}$.

Le rééquilibrage des charges

À la fin de l'exemple précédant, si un quatrième solveur a besoin d'un sous-problème, il n'y a plus de chemin de guidage disponible. Cela est détecté par les valeurs α, en effet, dans les trois derniers chemins de guidage de l'exemple, toutes les valeurs α sont mises à faux. Afin de pallier ce problème, un rééquilibrage des charges (technique aussi appelée *work stealing*) doit être mis en œuvre. Cela consiste à demander à un solveur de diviser son sous-problème ou problème courant en deux afin que ce quatrième solveur puisse aider dans la résolution. Cette manière de faire possède l'inconvénient d'exécuter trop souvent ce rééquilibrage des charges vers la fin de la recherche. En effet, quand les sous-problèmes deviennent de plus en plus facile, cette méthode engendre un grand nombre de divisions, cela est appelé l'effet *ping-pong*.

Exemple 4.2. La figure 4.9 est le résultat obtenu à partir de la figure 4.8 après l'arrivée de trois nouveaux solveurs (2, 3 et 4). l'exemple 4.1 explique l'introduction des solveurs 2 et 3. Afin d'avoir assez de sous-problèmes pour accueillir le solveur 4, un rééquilibrage des charges à lieu. Pour faire ce rééquilibrage, le solveur 1 choisit alors une variable dite de division via une heuristique. C'est la variable x_4 dans la figure 4.9. Puis deux nouveaux sous-problèmes sont générés à partir du chemin de guidage courant du solveur 1. Les solveurs 1 et 4 se positionnent alors sur ces deux nouveaux sous-problèmes, nous avons donc :

- le solveur 1 a le chemin $\{\,(x_1, \texttt{faux}\,), (x_2, \texttt{faux}\,), (\neg x_3, \texttt{faux}\,), (x_4, \texttt{faux}\,)\,\}$;

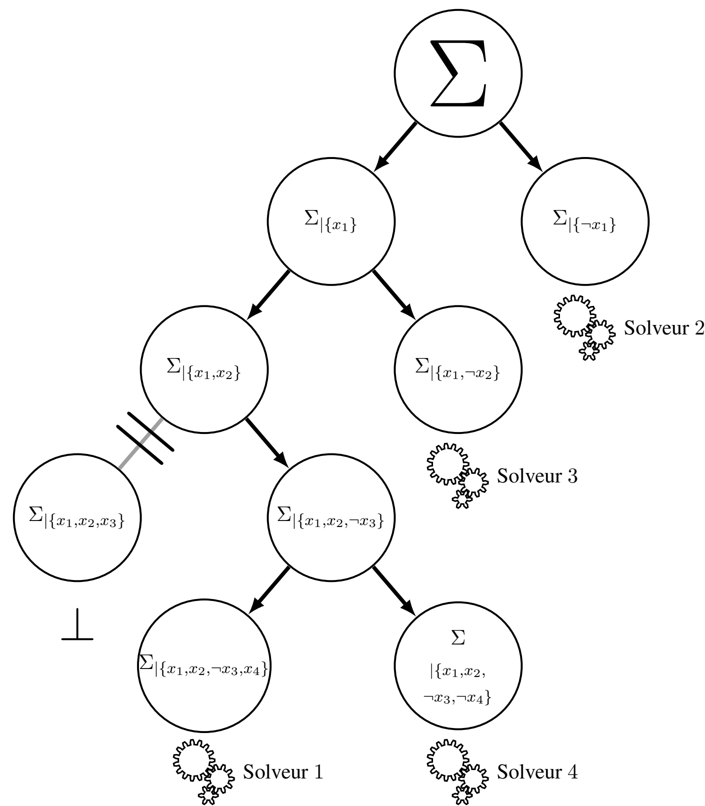


FIGURE 4.9 – La méthode « chemin de guidage ». À partir de la figure 4.8 et de l'exemple 4.1, un rééquilibrage des charges à lieu afin d'ajouter à la recherche le solveur 4.

- le solveur 2 a le chemin $\{\,(\neg x_1, \texttt{faux}\,)\,\}$;
- le solveur 3 a le chemin $\{\,(x_1, \texttt{faux}\,), (\neg x_2, \texttt{faux}\,)\,\}$.
- le solveur 4 a le chemin $\{\,(x_1, \texttt{faux}\,), (x_2, \texttt{faux}\,), (\neg x_3, \texttt{faux}\,), (\neg x_4, \texttt{faux}\,)\,\}$.

Les solveurs de type « chemin de guidage »

Après le solveur PSATO, certaines améliorations sont apparues, SATZ Jurkowiak *et al.* (2001) avec l'intégration de l'heuristique basée sur la propagation unitaire, l'échange de clauses a été intégré par PASAT Sinz *et al.* (2001). Citons aussi PMINISAT Chu *et al.* (2008) qui est la parallélisation du solveur MINISAT 2.0 (Sörensson et Eén 2008) avec la particularité d'exploiter les chemins de guidage des processus pour améliorer la qualité des clauses échangées.

4.2.2 MTSS : *Thread* riche et *threads* pauvres

▷ Vander-Swalmen, Dequen, et Krajecki (2008; 2009)
▷ Mémoire partagée

Le solveur MTSS a l'originalité d'employer deux approches différentes dans un solveur SAT parallèle : le *thread riche* et les *pauvres*. Le *thread riche* est associé à un solveur de type DPLL (section 2.2.3) tandis que les *threads pauvres* réalisent des traitements à plus petite échelle comme la propagation unitaire, le choix de la prochaine variable, l'apprentissage de clauses, ... Le *riche* suit une recherche ordonnée

dans l'arbre de recherche et possède donc une vue complète de la formule initiale. En revanche, les *threads pauvres* n'ont qu'une vue partielle de la formule initiale puisqu'ils traitent uniquement des sous-problèmes induits par des chemins de guidage. L'objectif est alors que les *threads pauvres* apportent des informations en provenance des sous-problèmes qu'ils traitent au *riche* avant que ce dernier arrive dans ces sous-espaces de recherche. En d'autres termes, le but des *threads pauvres* est de collecter des informations d'une manière prospective (*look-ahead*) afin d'améliorer l'efficacité dans la résolution du *riche*.

Afin de réaliser le concept innovateur *riche/pauvres*, les auteurs Vander-Swalmen *et al.* (2009) introduisent la notion d'arbre de guidage. Celui-ci représente la totalité des chemins de guidage via un arbre binaire et est partagé entre tous les *threads*. L'avantage est que la gestion de l'équilibrage des charges est naturellement gérée dans un tel arbre. En effet, le riche étant basé sur une recherche complète n'a pas besoin d'être équilibré et les *pauvres* équilibrent les tâches qui leurs sont destinées naturellement car celles-ci sont des traitements peu coûteux en temps de calcul.

Notons que sur certains points, les travaux présentés dans (Vander-Swalmen *et al.* 2009) ont des ressemblances avec l'une de nos contributions (chapitre 6). Celle-ci expose un solveur parallèle « diviser pour mieux régner » nommé AMPHAROS. En effet, dans AMPHAROS, un maître possède un arbre de guidage mais celui-ci n'est pas partagé avec tous les *threads*. De plus, les solveurs résolvent un sous-problème pendant un certain nombre de conflits. Une fois qu'un solveur atteint cette limite, il change de sous-problème. Cette manière de faire, permet aussi, comme dans (Vander-Swalmen *et al.* 2009), d'avoir un rééquilibrage des charges naturel. De plus, AMPHAROS permet de travailler en concurrence sur un sous-problème.

4.2.3 C-SAT : Plusieurs divisions en concurrence

▷ Ohmura et Ueda (2009)
▷ Mémoire distribuée : MPI

Le solveur C-SAT possède deux modes de fonctionnement : un mode *portfolio* et un mode « diviser pour mieux régner ». Ce dernier mode utilise les notions de chemin de guidage et de *work stealing* afin d'effectuer une division dynamique de l'espace de recherche. Comme chaque solveur rejoue la séquence de décision associée au chemin de guidage via la propagation unitaire au début de la résolution d'un sous-problème associé. Les chemins de guidage sont limités à une longueur donnée afin de ne pas ralentir la recherche.

L'originalité de ce solveur est d'effectuer plusieurs divisions dynamiques en même temps grâce à l'ajout d'un « Super-Maître » dans un modèle Maître/Esclave (figure 4.10). Les clauses apprises sont ainsi distribuées via le Maître et le Super-Maître. La détection des clauses redondantes est une opération coûteuse. Dans C-SAT, seul le Super-Maître essaye de réduire cette redondance.

4.2.4 SATCIETY : Du *peer-to-peer* (P2P)

▷ Schulz et Blochinger (2010)
▷ Mémoire distribuée

L'avantage du *peer-to-peer* est la transmission de données. Plus précisément, plus nous avons de machines disponibles, plus la vitesse de transmission d'un fichier est élevé. En effet, le *peer-to-peer* est basé sur la transmission de bits. Quand une machine reçoit quelques bits, elle est utilisée afin d'envoyer ces bits aux autres machines.

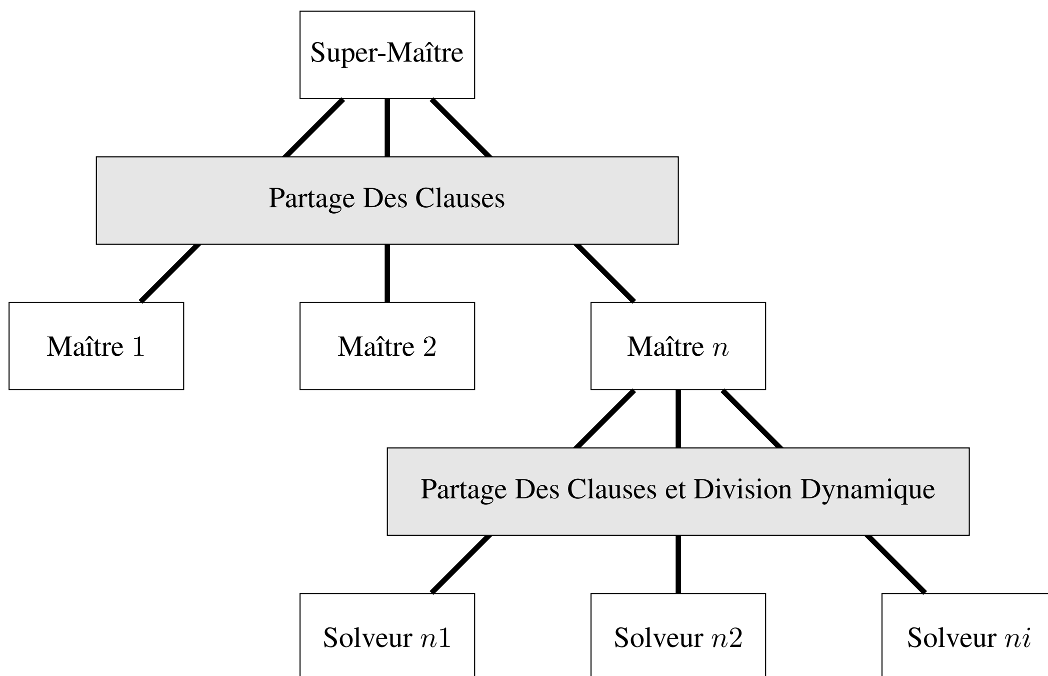


FIGURE 4.10 – Architecture du solveur C-SAT.

Comme PSATO, le solveur SATCIETY utilise aussi la notion de chemin de guidage. De plus, plusieurs étapes sont nécessaires afin de partager l'instance initiale avec les autres machines. Elles impliquent notamment une simplification/compression/décompression afin de modéliser le problème initial sous forme de bits. La phase de simplification est faite grâce au pré-processeur SATELITE.

La compression est faite grâce à un encodeur nommé BCNF. Il examine le nombre de variables (apporté dans le préambule d'un fichier DIMACS) afin de calculer le nombre de bits $|\ell|_{\text{BCNF}}$ nécessaire à l'encodage d'un seul littéral :

$$|\ell|_{\text{BCNF}} = partieEntiere(log_2|\mathcal{V}|) + 1$$

Par rapport aux autres solveurs de l'état de l'art, SATCIETY est décentralisé (cela est directement induit par le *peer-to-peer*). Rappelons que cela signifie que toutes les machines échangent directement les informations. À l'inverse, dans une configuration où l'échange est centralisée, les informations sont partagées via une autre machine appelée maître ou serveur.

Pour finir, remarquons que les échanges de clauses apprises sont réalisées via UDP et XMPP (protocole XML transmis via TCP/IP). De plus, les auteurs réalisent ce partage via la notion de voisin (comme pour le solveur TOPOSAT, section 4.1.10). Entre deux machines voisines, les échanges sont réalisés dynamiquement en fonction des tailles et du nombre de clauses possédé par l'envoyeur et le receveur.

4.2.5 PART-TREE-LEARNING : L'échange des clauses

▷ Hyvärinen, Junttila, et Niemelä (2011)

▷ Mémoire distribuée

Définition 4.8 (*Assumption*)**.** Un chemin de guidage peut être présenté sous forme d'une ***assumption***. Une *assumption* (hypothèse) est définie comme un ensemble de littéraux supposé être vrai. Cet ensemble peut être vu comme un cube (une conjonction de littéraux) et la recherche peut être restreinte à ce cube.

Exemple 4.3. Soit Σ la formule initiale, le solveur 1 de la figure 4.9 est en train de résoudre le sous-problème représenté par la formule $\Sigma \wedge \underline{x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4}$. Dans celle-ci, l'*assumption* est soulignée.

Les auteurs Hyvärinen *et al.* (2011) simplifient les sous-problèmes directement en fonction de leurs *assumptions*. Par conséquent, ils ne peuvent plus faire la différence entre les clauses apprises étant des conséquences logiques de la formule initiale et celles étant des conséquences logiques d'un sous-problème particulier. Notons que dans nos contributions, nous ne simplifions pas directement un sous-problème avec son *assumption*, nous décidons simplement les littéraux de cette *assumption* au début de la recherche comme si ils étaient des décisions provenant de l'heuristique de choix de variable. Cela permet de garder la trace des clauses apprises pendant la recherche. En revanche dans Hyvärinen *et al.* (2011), les clauses véritablement conséquences logiques de la formule doivent être détectées. Cela est réalisé en marquant les clauses (alors considérées comme « confiantes ») de la formule initiale et celles provenant des résolutions n'impliquant pas de littéraux inclus dans l'*assumption* du sous-problème associé.

Exemple 4.4. Soient le sous-problème $\Sigma \wedge \neg x_1$ et la formule initiale $\Sigma = \{(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)\}$. Dans Hyvärinen *et al.* (2011), le sous-problème est représenté par la formule $\{(x_1 \vee x_2 \vee x_3)^s \wedge (x_2 \vee \neg x_3)^s \wedge \neg x_1\}$ où les clauses considérées comme étant « confiantes » sont marquées par s. Décider le littéral $\neg x_2$ entraîne un conflit générant la clause apprise x_2. Comme cette clause a été générée via des résolutions de clauses n'étant pas toutes marquées par s ($\neg x_1$), la clause apprise x_2 n'est pas considérée « confiante » et n'est donc pas échangée.

4.2.6 CUBEANDCONQUER : Une décomposition statique

▷ Heule, Kullmann, Wieringa, et Biere (2012)
▷ Mémoire distribuée : Script

La méthode *Cube And Conquer* introduite par Heule *et al.* (2012) vise à réduire le temps de résolution des instances difficiles. Cette approche est composée de deux phases.

La première appelée *Cube* consiste à diviser le problème original en un très grand nombre de sous-problèmes (alors appelés cubes) via un solveur *look-ahead*. La figure 4.11 représente une telle division sous la forme d'un arbre binaire. Dans celle-ci, les sous-problèmes sont représentés par les feuilles avec des engrenages. En revanche, les feuilles représentant des croix sont des sous-problèmes prouvés insatisfaisables pendant la phase *Cube*.

La deuxième phase appelée *Conquer* consiste à résoudre chaque sous-problème (cube) grâce à plusieurs solveurs CDCL distincts (feuilles avec des engrenages dans la figure 4.11).

L'intuition de *Cube And Conquer* est basée sur la force d'un solveur *look-ahead*. En utilisant une heuristique globale *look-ahead*, celui-ci effectue alors des décisions plus précises qu'un solveur CDCL, mais plus coûteuses. L'idée est alors d'utiliser un solveur *look-ahead* uniquement sur la division du problème en sous-problèmes. Cette méthode a donc pour but de passer d'une résolution *look-ahead* à une résolution CDCL de type *look-back* quand un des sous-problèmes semble devenir facile. Elle a l'avantage d'être parallèle par sa nature.

Exemple 4.5. Soit Σ une formule sous forme CNF représentant notre instance initiale. La phase *cube* représentée par la figure 4.11 a créé quatre sous-problèmes (cubes). Chaque cube est représenté par le chemin de la racine à une feuille (avec un engrenage). Chaque sous-problème Σ_n est alors représenté par la formule initiale Σ accompagnée d'un cube (conjonction de littéraux). La figure nous donne ainsi quatre sous-problèmes tels que :

- $\Sigma_1 = \Sigma \wedge (x_2 \wedge x_3 \wedge x_1)$

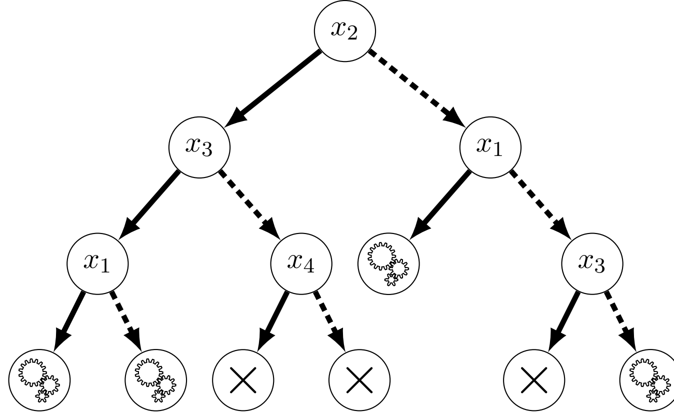


FIGURE 4.11 – Méthode parallèle *Cube And Conquer* (solveur CUBEANDCONQUER). L'arbre représente une division du problème initial en sous-problèmes (phase *Cube*). Une flèche pleine (resp. en pointillé) signifie que la variable associée est affectée à `vrai` (resp. `faux`). Les feuilles représentant des engrenages sont des sous-problèmes à résoudre (Phase *Conquer*) tandis que celles représentant des croix sont des sous-problèmes déjà prouvés insatisfaisables pendant la division (la phase *Cube*).

- $\Sigma_2 = \Sigma \wedge (x_2 \wedge x_3 \wedge \neg x_1)$
- $\Sigma_3 = \Sigma \wedge (\neg x_2 \wedge x_1)$
- $\Sigma_4 = \Sigma \wedge (\neg x_2 \wedge \neg x_1 \wedge \neg x_3)$

Algorithme

Un solveur *look-ahead* (MARCH) est modifié afin d'en faire un outil de partitionnement du problème initial. Basé sur un solveur DPLL, il effectue un parcours en profondeur d'abord avec *backtrack* en utilisant une heuristique *look-ahead*.

Algorithme 4.1 : CUBEPHASE

Données : Σ la formule initiale, $\mathcal{A}$ l'ensemble de cubes, $\mathcal{C}$ l'ensemble de clauses apprises, l'ensemble des littéraux de décisions $\mathcal{L}_d$ et l'ensemble des littéraux impliqués via la propagation unitaire $\mathcal{L}_i$.

Résultat : $\mathcal{A}$ l'ensemble des cubes devant être envoyés à des solveurs CDCL.

```
1  Début
2    (Σ,L_i) ⟵ simp(Σ,L_d,L_i);
3    si falsifie(L_d,L_i) alors                              /* Un conflit */
4      retourner (A,C ∪ {¬L_d});                     /* Apprend une clause */
5    si coupureHeuristique() alors
6      retourner (A ∪ {L_d},C);                           /* Ajoute le cube */
7    ℓ ⟵ decisionHeuristique(Σ,L_d,L_i);
8    (Σ,C) ⟵ CUBEPHASE(Σ,A,C,L_d ∪ {ℓ},L_i);
9    retourner CUBEPHASE(Σ,A,C,L_d ∪ {¬ℓ},L_i);
10 Fin
```

L'algorithme 4.1 représente la phase *cube* du solveur CUBEANDCONQUER. Pour chaque nœud de l'arbre de recherche, la propagation unitaire est effectuée ainsi que d'autres simplifications, cela peut

impliquer d'autres littéraux (ligne 2). Ensuite, l'algorithme vérifie s'il y a un conflit (ligne 3). Dans ce cas, une clause est apprise afin de retenir ce conflit. Remarquons qu'il ne s'agit pas ici d'une analyse de conflit comme dans un solveur CDCL. La clause apprise est simplement représentée par le complément du cube représentant les décisions de la racine aux feuilles. Après un conflit, l'algorithme termine la récurrence (ligne 4).

Par la suite, une heuristique dite de coupure est en charge de décider si il est temps de créer un sous-problème ou si le sous-problème en cours doit encore être divisé. En d'autre termes, l'algorithme choisit si on peut envoyer le cube à un solveur CDCL (ligne 5). Si l'heuristique accepte le sous-problème, celui-ci est retenu par le cube associé dans l'ensemble $\mathcal{A}$ puis la récurrence se termine (ligne 6).

Pour finir, l'algorithme fait deux appels récursifs comme DPLL (ligne 8 et 9). Pour cela, il choisit un littéral « de division » via une heuristique *look-ahead* (ligne 7). À la fin de la procédure, l'ensemble des clauses $\mathcal{C}$ et l'ensemble des cubes $\mathcal{A}$ sont simplifiés. Les solveurs CDCL reçoivent les sous-problèmes tel que $\Sigma \wedge (a \in \mathcal{A}) \wedge \mathcal{C}$, soit la formule accompagnée d'un cube avec quelques clauses apprises. Comme nous le montre l'exemple suivant, les clauses de $\mathcal{C}$ forceront dans certains cas quelques variables, que nous pouvons donc enlever de divers cubes.

Exemple 4.6. Toujours via la figure 4.11, les deux premiers conflits représentés par une croix sont les complémentaires des décisions de la racine à leurs feuilles, on a donc les clauses apprises $\neg x_2 \vee x_3 \vee \neg x_4$ et $\neg x_2 \vee x_3 \vee x_4$. À la fin de l'algorithme, une optimisation effectue une résolution sur ces deux clauses pour en faire une seule : $\neg x_2 \vee x_3$. Ainsi quand x_2 sera affectée, la clause propagera x_3. On peut donc enlever la variable x_3 des cubes qui contiennent la variable x_2 sans oublier d'envoyer cette clause apprise aux solveurs CDCL. Cela nous donne :

- $\Sigma_1 = \Sigma \wedge (x_2 \wedge x_1) \wedge \mathcal{C}$
- $\Sigma_2 = \Sigma \wedge (x_2 \wedge \neg x_1) \wedge \mathcal{C}$
- $\Sigma_3 = \Sigma \wedge (\neg x_2 \wedge x_1) \wedge \mathcal{C}$
- $\Sigma_4 = \Sigma \wedge (\neg x_2 \wedge \neg x_1 \wedge \neg x_3) \wedge \mathcal{C}$

Remarque 4.1. Bien que ces clauses apprises soient utiles surtout lorsque que celles-ci sont simplifiées, les auteurs signalent que l'envoi de ces clauses apprises à un solveur CDCL est optionnel. En effet, le sous-problème généré étant différent du problème initial, il y a peu de chance que les clauses apprises près de la même branche (celle formant le cube) servent réellement à ce sous-problème. Néanmoins, une clause apprise d'une autre branche (une branche totalement différente du cube) a plus de chance de servir à d'autres sous-problèmes, surtout si sa taille est petite.

L'heuristique de coupure

Les auteurs ont implémenté trois heuristiques différentes pour sélectionner les variables divisant au mieux l'arbre. C'est la dernière qui s'est révélée être la plus efficace.

La méthode A propose de couper les branches après un certain nombre de décisions k fixé à l'avance. L'avantage de cette heuristique est de savoir quel est le nombre maximal de cubes (2^k sous-problèmes). Elle possède néanmoins un inconvénient majeur : comme les cubes sont coupés au même niveau de l'arbre de recherche *look-ahead*, ceux-ci sont disproportionnés au niveau de leur difficulté. En effet, certains cubes seront très faciles à résoudre tandis que d'autres seront beaucoup plus difficiles.

La méthode B propose de couper les branches dès qu'on a dépassé un certain pourcentage de variables assignées. Cette heuristique pose principalement problème dans le fait que pour certaines instances, le nombre de sous-problèmes générés est trop élevé alors que pour d'autres, ils ne sont pas assez nombreux.

La méthode C est l'heuristique intégrée dans le solveur CUBEANDCONQUER. Elle compare le produit du nombre de décisions et du nombre de littéraux impliqués avec le nombre total de variables dans l'instance. Ce dernier nombre est affiné grâce à une variable dynamique α. La branche est coupée à la condition de respecter cette inéquation :

$$nb_{dec} * nb_{dec \cup imp} > \alpha * nb_{var}$$

avec :

- nb_{dec} est le nombre de variables décidées ;
- $nb_{dec \cup imp}$ est le nombre de variables de décisions plus le nombre de variables impliquées ;
- nb_{var} est le nombre de variables de la formule initiale ;
- α est une variable dynamique qui est initialisée à 1000.

À chaque appel, la variable α est augmentée de 5 pour empêcher qu'elle ne devienne pas trop basse. Puis lors d'un conflit, elle est diminuée de 15 afin d'essayer de couper les branches voisines avant l'apparition d'autres conflits. De ce fait, plus il y a de conflits, plus l'algorithme génère des sous-problèmes (cubes). Cela permet aussi de ne pas trop entrer en profondeur dans l'arbre de recherche. À partir d'un certain moment en fonction du nombre de décisions, on abaisse toujours la variable α afin de couper toutes les branches et de créer tous les cubes en quelques secondes. Cela permet de ne pas passer trop de temps dans la phase *cube*.

L'heuristique de choix de variable

Pour l'heuristique de choix de variable *look-ahead*, la phase *cube* se base sur la fonction `eval`(ℓ) comptant le nombre de littéraux assignés pendant la propagation du littéral ℓ. Le solveur CUBEANDCONQUER choisit ensuite la variable maximisant `eval`$(\ell) \times$ `eval`$(\neg\ell)$, les égalités sont brisées par la valeur obtenue par `eval`$(\ell) +$ `eval`$(\neg\ell)$. Cette heuristique possède deux avantages. Le premier est qu'elle est peu coûteuse comparée à une heuristique lourde en calcul, comme celles basées sur les clauses réduites mais non satisfaites. Par conséquent, elle ne requière pas une structure de données complexe comme celles souvent implémentées dans les solveurs *look-ahead*. Le deuxième avantage est que cette heuristique est meilleure sur des instances qui contiennent beaucoup de clauses binaires. Les auteurs précisent que cela est souvent le cas des instances industrielles. En effet, on peut remarquer que plus une instance contient des clauses binaires, plus l'ensemble d'arrivée de la fonction `eval`(ℓ) sera diversifié. En contrepartie, quand il n'y a pas ou peu de clauses binaires, cela réduira considérablement l'efficacité de cette heuristique.

La résolution des cubes

Les auteurs se reposent sur un outil de parallélisation appelé TARMO. La technique *multijob* utilisée pour CUBEANDCONQUER consiste à attribuer un job (représentant un cube ou un sous-problème) dès qu'un nœud de calcul est au repos. Quand deux nœuds sont au repos en même temps, l'ordre d'assignement des cubes est indéfini, mais il est garanti que deux nœuds ne fonctionneront jamais sur le même

cube. Les auteurs ont aussi expérimenté une autre stratégie appelée *multijob+* qui se base sur *multijob* excepté qu'il peut assigner un cube qui est déjà en train d'être résolu par d'autres nœuds de calculs si ceux-ci seraient de toute façon restés au repos. Cela découle du fait que certains cubes sont très faciles et se résolvent donc rapidement tandis que d'autres sont trop longs à résoudre. Grâce à cette amélioration, une fois que les cubes les plus simples ont été résolus, tous les autres, plus difficiles, peuvent donc être en cours de résolution plusieurs fois. Pour finir, remarquons que le solveur CUBEANDCONQUER est la base du solveur TREENGELING, ce dernier à remporté de nombreuses médailles dans les compétitions.

4.2.7 DOLIUS : Une API simple et claire

▷ Audemard, Hoessen, Jabbour, et Piette (2014c)
▷ Mémoire distribuée : Socket

Le solveur DOLIUS est un des solveurs basé sur la notion de chemin de guidage. Dans celui-ci, la séparation en deux sous-problèmes lors du rééquilibrage des charges est réalisée sur la variable du solveur concerné ayant la valeur EVSIDS la plus élevée. Ce solveur permet d'étudier plusieurs manières d'échanger les clauses apprises et propose à cette fin une API permettant de reprogrammer plus facilement de nouvelles techniques. Dans la version proposée par les auteurs, DOLIUS est basée sur le solveur *portfolio* PENELOPE et échange les clauses possédant une valeur LBD inférieure ou égale à 15. Notons que dans ce solveur, l'échange des clauses peut se faire d'une manière plus simple que dans le solveur PART-TREE-LEARNT. Le solveur simplifie un sous-problème en considérant chaque littéral du cube associé comme un niveau de décision. Cela permet donc de considérer toutes les clauses apprises d'un sous-problème comme les clauses apprises du problème initial.

4.3 Conclusion

De nombreux solveurs sont aujourd'hui distribués afin de permettre l'utilisation de plusieurs machines. Néanmoins, la différence de vitesse entre les différentes manières (mémoire partagée *versus* distribuée) d'échanger les informations doit être prise en compte.

Dans chacune des deux approches (*portfolio* et « diviser pour mieux régner »), le partage d'informations tel que les clauses apprises permet d'améliorer considérablement l'efficacité des solveurs parallèles. Toutefois, la quantité de données doit être contrôlée d'une manière heuristique afin de ne pas encombrer à la fois les solveurs (la propagation unitaire) et le réseau.

Dans la prochaine partie, nous exposons nos contributions. Dans celle-ci, nous présentons plusieurs approches améliorant l'efficacité des méthodes actuelles. La première contribution (chapitre 5) expose quelques manières de faire des décompositions statiques. Par la suite, nous avons implémenté un solveur nommé AMPHAROS afin de comparer des décompositions statiques à d'autres dynamiques. Cette contribution a été un levier permettant de créer un solveur « diviser pour mieux régner » et employant de nouvelles techniques (chapitre 6). Ce solveur, nommé AMPHAROS à la particularité d'échanger des informations propres aux sous-problèmes. De plus, il permet aussi de résoudre un sous-problème en concurrence par plusieurs unités de calcul grâce à des heuristiques dédiées. Pour finir, notre dernière contribution est basée sur les modèles de programmation parallèle. Notamment, via un solveur nommé D-SYRUP (chapitre 7), nous montrons que ces modèles ont un impact considérable sur les performances d'un solveur distribué. De ce constat, nous apportons un modèle de programmation original permettant d'obtenir des meilleures performances en envoyant les clauses sur le réseau dès que possible.

Contributions

Chapitre 5

De la décomposition statique à celle dynamique

Sommaire

5.1 Décomposition via les bandits manchots : UCTSAT 117
5.1.1 Utilisation d'UCT pour générer des cubes 118
5.1.2 Algorithme 119
5.1.3 Descente 121
5.1.4 Sélection 125
5.1.5 Simulation 127
5.1.6 Remontée 127
5.1.7 Récupération des cubes 127
5.2 Décomposition via CUBEANDCONQUER : SWARMSAT 128
5.2.1 Les différents processus 128
5.2.2 Communication 129
5.2.3 Mode de fonctionnement 129
5.3 Expérimentations 130
5.3.1 UCTSAT 130
5.3.2 SWARMSAT 131
5.4 Conclusion 132

Le point de départ des travaux de cette thèse est présenté dans ce chapitre. Ces travaux ont donné lieu à une publication nationale (Audemard *et al.* 2015) et induisent des décompositions statiques et dynamiques dans une approche « diviser pour mieux régner ». Pour rappel, dans une telle approche, une décomposition statique est une division du problème en sous-problèmes avant leur résolution. En revanche, dans une décomposition dynamique, des sous-problèmes sont générés pendant la résolution du problème initial ou d'autres sous-problèmes (voir la section 4.2 pour plus de précisions sur les décompositions).

Nous commençons par présenter une approche de décomposition statique et proposons une méthode nommée UCTSAT (section 5.1). Ce dernier permet de traiter le problème de diversification *Versus* intensification de la recherche. Plus précisément, nous utilisons l'algorithme UCT (Kocsis et Szepesvári 2006) (heuristique UCB (*Upper Confidence Bounds*) appliqué à un arbre) pour générer les sous-problèmes dans une méthode « diviser pour régner ». Ensuite, nous présentons un solveur modulaire collaboratif, nommé SWARMSAT (section 5.2), basé sur une résolution massivement parallèle de cubes générés selon la méthode proposée dans CUBEANDCONQUER (voir section 4.2.6, page 109, Heule *et al.* (2012)). Notons que ce solveur est capable de faire plusieurs sortes de décompositions : certaines statiques et d'autres dynamiques (section 5.2.3). Dans la partie expérimentation (section 5.3), nous étudions l'impact des différents composants de SWARMSAT (qualité des cubes, coût des communications, . . .). Puis, nous comparons SWARMSAT avec un solveur collaboratif de l'état de l'art nommé DOLIUS (Audemard *et al.* 2014c) et nous discutons des axes d'améliorations possibles.

5.1 Décomposition via les bandits manchots : UCTSAT

Pour rappel, un sous-problème est représenté par un cube qui est une conjonction de littéraux. Un sous-problème peut donc aussi se voir comme une interprétation partielle. Dans cette approche, nous devons faire une recherche arborescente afin de trouver plusieurs interprétations partielles représentant les sous-problèmes et cela d'une manière statique. Nous utilisons la notion d'*assumption* (définition 4.8, section 4.2.5, page 108) afin de résoudre les cubes créés après leur génération.

De ce fait, les cubes générés doivent être sélectionnés d'une manière heuristique afin de diviser le problème de telle sorte que les sous-problèmes soient assez équilibrés. En effet, dans une décomposition statique, l'équilibrage des charges ne peut pas être réalisé comme pour une décomposition dynamique. Plus précisément, nous ne pouvons pas, dans cette méthode statique, faire comme dans les techniques de *work stealing* et de chemin de guidage (voir section 4.2.1, page 104). Pour le lecteur intéressé, le problème de l'équilibrage des charges est exposé d'une manière générale dans la section 3.5.2 (page 76) de l'état de l'art. Dans une telle méthode, pour prouver la satisfaisabilité d'une instance, il suffit qu'un des cubes (sous-problèmes) soit satisfaisable. A contrario, pour prouver l'insastifaisabilité d'une instance, il faut montrer que tous les cubes sont insatisfaisables. Par conséquent, pour les instances satisfaisables, le but est alors d'avoir beaucoup de cubes facilement satisfaisables. En revanche, pour celles insatisfaisables, les cubes sont forcément insatisfaisables, mais le but est donc que ces cubes soient plus faciles à résoudre que le problème initial.

Plusieurs techniques proposées dans la littérature peuvent nous donner des idées sur la conception algorithmique d'une telle méthode statique (chapitre 4). Notamment, le solveur CUBEANDCONQUER génère des cubes exactement comme nous pourrions le souhaiter à un petit détail près, cet algorithme n'obtient pas d'ordre sur la difficulté des cubes. Plus précisément, nous ne pouvons savoir si un cube généré (sous-problème) est plus facile qu'un autre (section 4.2.6). Notre objectif est donc d'avoir une méthode qui est capable de nous renvoyer un tel ordre des sous-problème.

Dans un premier temps, nous allons expliquer le choix d'UCT afin de générer des cubes et appelons cette méthode UCTSAT. Ensuite, après une description de la structure utilisée dans UCTSAT, nous expliquons, étape par étape, l'algorithme qui en découle avec ses subtilités. Par la suite, nous parlons des heuristiques utilisables pour UCTSAT et celles implémentées. Pour finir, nous concluons et étudions les améliorations possibles.

5.1.1 Utilisation d'UCT pour générer des cubes

Dans l'optique de choisir les « meilleurs » cubes dans un arbre de recherche, nous avons été confronté au dilemme exploitation/exploration. Devons nous exploiter un cube déjà considéré comme « bon » en lui ajoutant des littéraux, c'est-à-dire, en développant d'autres sous-problèmes à partir du sous-problème considéré par ce cube ? Ou devons nous explorer d'autres sous-problèmes, c'est-à-dire, retenter notre chance avec un nouveau cube ?

L'algorithme UCT, initialement prévu pour les jeux à deux joueurs et les bandits manchots (définition 4.6, page 96) traite ce problème. Il considère le dilemme exploitation/exploration de cette manière : exploiter un état de la partie qui semble être bon, ou explorer d'autres états moins bons. Ce dilemme est très connu, il a été abordé par l'algorithme UCT dans de nombreux domaines (Gelly et Wang 2006, Silvan 2012, Senseis 2014). Notamment, Cet algorithme a prouvé son efficacité en permettant pour la première fois à une intelligence artificielle de battre un joueur humain professionnel pour le jeu de Go (CMAP *et al.* 2006).

L'une des particularités d'UCT est qu'il induit plusieurs simulations (expériences), celles-ci sont coûteuses et difficilement adaptables au problème SAT. Concrètement, ces simulations sont effectuer afin d'essayer d'obtenir un gain : la probabilité d'un évènement. Par exemple, une simulation possible et adaptée à SAT est de lancer un solveur CDCL sur un sous-problème pendant un certain laps de temps afin qu'il nous donne des informations sur le sous-problème en question comme par exemple, le nombre de littéraux propagés ou de conflits. Deux raisons essentielles nous ont poussé à choisir UCT afin de l'adapter à une recherche de cubes dans UCTSAT. La première est qu'il est capable de traiter d'une manière efficace le dilemme exploration/exploitation que nous retrouvons au niveau de la création des cubes. La deuxième intuition provient du fait que l'algorithme UCT est utilisé dans des problèmes possédant un immense espace de recherche, comme pour le problème SAT.

5.1.2 Algorithme

Remarque importante, l'algorithme UCTSAT génère des sous-problèmes, il ne les résout pas. Nous distinguons deux sortes de solveurs (deux *workers*) dans UCTSAT, autrement-dit, nous avons deux arbres de recherche différents en coopération.

Le premier, est un solveur SAT de type CDCL pouvant revenir à la racine de son arbre de recherche, via un redémarrage (section 2.3, page 34). Le second est un arbre n-aire représentant un algorithme UCT que nous avons adapté à SAT. Celui-ci représente une génération de cubes dans UCTSAT. Ces deux arbres sont étroitement liés via les cubes. Concrètement, celui représentant UCT garde les sous-problèmes courants en mémoire tandis que le solveur SAT les simule (les teste).

Dans la suite de ce manuscrit, quand aucune précision est apportée sur la nature de l'arbre, nous considérons que nous parlons de l'arbre n-aire gardant les cubes en mémoire et pas de l'arbre induit par un solveur SAT lors des simulations.

UCTSAT se divise en quatre étapes (figure 5.1) :

- La *descente* dans l'arbre de recherche (dans un sous-problème). Son but est de descendre dans les nœuds de l'arbe suivant une heuristique ;
- La *sélection* d'une feuille (d'une variable). Une fois sur une feuille de l'arbre de recherche grâce à la descente, son but est de sélectionner un nouveau état pour créer un nouveau nœud.
- La *simulation* d'un nœud de l'arbre (simulation d'un sous-problème). Son but est de juger le nouveau nœud d'une manière heuristique via plusieurs expériences. Ces expériences renvoient alors un gain ;
- La *remontée* dans l'arbre de recherche. Une fois que le gain attribué au nouveau nœud, la remontée consiste à mettre à jour les gains des nœud parents en fonction du gain du nouveau nœud.

Ces étapes sont réitérées dans cet ordre à plusieurs reprises suivant un nombre d'itérations UCT que nous désirons appliquer. Chacune de ces étapes a été adaptée à SAT afin de générer les cubes demandés. Après quelques informations globales sur la structure de données à l'intérieur d'UCTSAT, nous allons voir les remaniements que nous avons dû réaliser sur l'algorithme de base UCT afin de pouvoir l'utiliser dans le cadre de SAT.

Chaque nœud de l'arbre représente un littéral. Les nœuds, excepté la racine, contiennent distinctement trois informations essentielles : la première est un littéral dû au problème SAT tandis que les deux autres proviennent de l'algorithme UCT. L'un est la moyenne pondérée d'un nœud par rapport à ses fils (*average*) alors que l'autre est le nombre de passages dans un nœud par l'algorithme UCTSAT (*passing*). En plus de ces attributs, chaque nœud possède deux méthodes utilisées dans l'étape de descente :

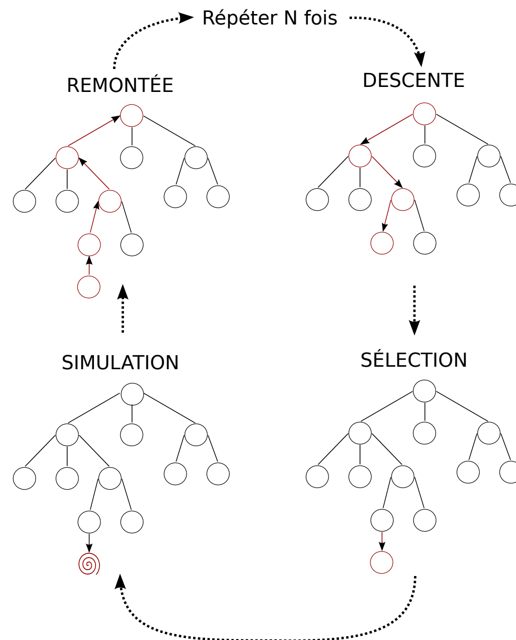


FIGURE 5.1 – Étapes de l'algorithme UCT.

`calculUCB()` et `calculUCBspéciale()` (ces méthodes sont présentées dans la suite de ce chapitre). Ces deux méthodes sont décrites dans les prochaines sous-sections. Par contre, le nœud racine possède uniquement les attributs *average* et *passing*, ainsi qu'une seule méthode : `calculUCB()`. En effet, la racine ne représente pas un littéral. De ce fait, certaines fonctionnalités et informations ne sont pas utiles. La figure 5.2 représente un aperçu de l'arbre ainsi que la structure des nœuds après plusieurs itérations UCT.

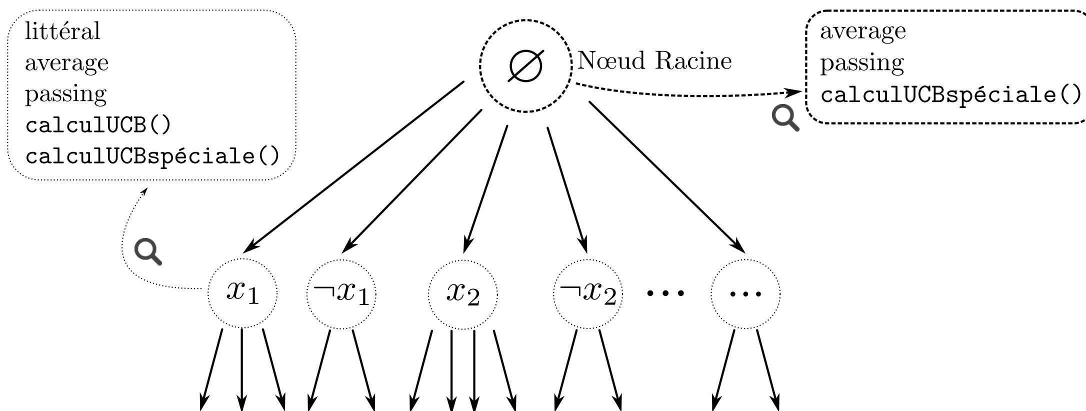


FIGURE 5.2 – Structure des nœuds dans UCTSAT.

Contrairement à l'algorithme UCT de base, qui considère l'ensemble des états atteignables à chaque ouverture de nœud, UCTSAT ajoute des nœuds à un père en considérant une variable x. De ce fait, nous créons deux nœuds à chaque ajout d'une telle variable. L'un représentant le littéral x et l'autre, le littéral opposé $\neg x$. Cette solution a été choisie afin de limiter l'explosion de la taille de l'arbre considéré (voir la section 5.1.4 de l'étape de sélection). Le fait que l'arbre soit n-aire provient de la descente qui peut

choisir un nœud contenant déjà des fils. À présent, nous allons expliquer chaque étape.

5.1.3 Descente

Algorithme 5.1 : `descenteUctSATV1` ($\mathcal{R}$)

```
    Données : R la racine de l'arbre MANAGER
    Résultat : Le nœud utilisé pour l'étape de la sélection
1   Début
2       tant que possèdeFils(R) faire
3           si maxUCBSpécial(R) alors
4               retourner R;
5           SolveurSATdécide(R);
6           R ⟵ filMaxUCB(R);
7           passing(R) ⟵ passing(R) + 1;
8       retourner R;
9   Fin
```

La descente, première étape provenant d'UCT a pour but de descendre dans l'arbre de recherche en choisissant pour chaque nœud parcouru, la branche maximisant les valeurs UCB de ses fils (via `calculUCB()`) ou une valeur UCB spéciale (via `calculUCBspéciale()`). L'algorithme 5.1 représente une première version de la descente que nous améliorerons dans la suite de ce chapitre. Tant que le nœud courant n'est pas une feuille (ligne 2), l'algorithme descend dans le fils possédant la valeur UCB (`calculUCB()`) la plus grande (ligne 6). Cependant, nous avons deux cas d'arrêts :

- le premier est si la valeur UCB spéciale est plus grande que toutes les valeurs UCB des fils (ligne3), alors nous retournons le nœud en cours ;
- tandis que le second arrêt est réalisé une fois que nous sommes arrivés sur une feuille de l'arbre.

À chaque fois que nous descendons dans un nœud, un solveur SAT associé à UCTSAT décide le littéral en question (ligne 5). De plus, ce solveur effectue aussi les propagations unitaires possibles. À présent, nous allons voir sur un exemple concret les deux cas d'arrêt via la figure 5.3. Cette figure se décompose en deux sous-figures : à gauche, nous retrouvons le retour normal (ligne 8 de l'algorithme 5.1) sur une feuille tandis qu'à droite, nous avons le retour à l'intérieur de la boucle (ligne 4) via la valeur UCB spéciale. Cette valeur spéciale permet d'avoir un arbre n-aire car elle arrête la descente sur un nœud qui contient déjà des fils.

Exemple 5.1. La sous-figure à gauche (de la figure 5.3) nous montre une situation où la valeur UCB spéciale n'est jamais plus grande que les valeurs UCB des fils. Sur cette sous-figure, nous comparons la valeur UCB spéciale (0.106) avec les valeurs UCB des fils (0.113, 0.112, 0.115, 0.126, 0.090, 0.106) de la racine. La plus grande valeur étant celle du nœud possédant le littéral $\neg x_2$ (0.126), nous réitérons la descente sur ce nœud qui n'est pas une feuille. Itérativement, nous continuons de descendre dans les nœuds représentés par les littéraux x_6, x_3 et $\neg x_5$. Ce dernier étant une feuille, la descente renvoie ce nœud. Pendant cette descente, le solveur SAT associé décide donc les littéraux $\neg x_2$, x_6, x_3 et $\neg x_5$.

La seconde sous-figure de droite nous montre le cas d'arrêt via la valeur UCB spéciale. Comme précédemment, la plus grande valeur étant celle du nœud possédant le littéral $\neg x_2$ (0.126), nous atteignons ce nœud. Par la suite, nous comparons la valeur UCB spéciale (0.312) avec les valeurs UCB des fils (0.712,0.678) du nœud $\neg x_2$. Le nœud représentant le littéral x_6 (0.712) est donc choisi. Toujours de la

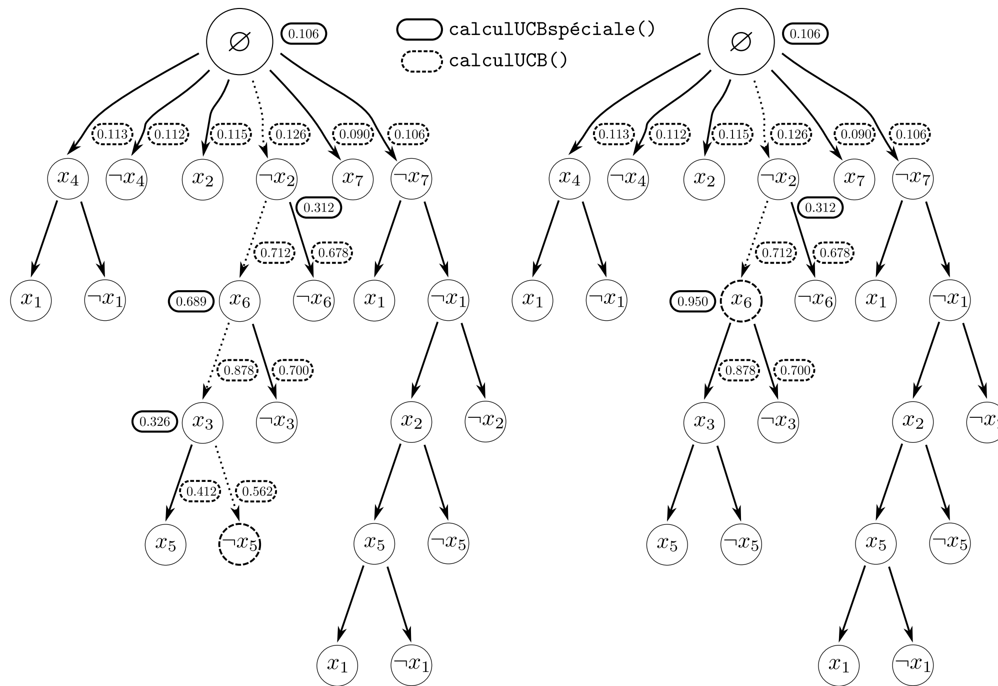


FIGURE 5.3 – La descente dans UCTSAT.

même manière, nous comparons la valeur UCB spéciale du littéral x_6 (0,950) avec les valeurs UCB des fils (0.878,0.700). Cette fois, c'est la valeur UCB spéciale qui est la plus grande. L'algorithme renvoie alors le nœud représenté par le littéral x_6. De plus, le solveur SAT associé décide donc les littéraux $\neg x_2$ et x_6.

Valeurs UCB

Pour bien comprendre UCB dans UCTSAT, nous exposons les valeurs UCB spéciales après avoir expliqué celles qui ne sont pas spéciales. Les valeurs UCB de chaque nœud sont calculées lors de la décente. Plus précisément, lorsque l'algorithme est arrivé à un nœud, il calcule les valeurs UCB des nœud fils. Nous choisissons de calculer les valeurs UCB des fils via la formule mathématique d'UCB, cela est réalisé par la méthode `calculUCB()` propre à un nœud. Une adaptation possible de cette formule UCB à SAT pour nos nœuds dans l'arbre de recherche est la suivante. Lorsque `passing` > 0, l'indice UCB associé au nœud i est :

$$\text{UCB}_i = \underbrace{\texttt{average}_i}_{\text{Terme Exploitation}} + \underbrace{\mathcal{K} * \sqrt{\frac{\ln(\texttt{passing}_{pere(i)})}{\texttt{passing}_i}}}_{\text{Terme Exploration}}$$

Exemple 5.2. La figure 5.4 représente la branche à choisir pour descendre dans l'arbre : le nœud qui a le plus grand UCB. Pour chaque fils i du nœud racine, la valeur UCB_i est calculée suivant ce tableau en prenant $\mathcal{K}$ à 0.009. De plus, $\texttt{passing}_{pere(i)} = 50$ est le nombre d'itérations UCT à cet instant.

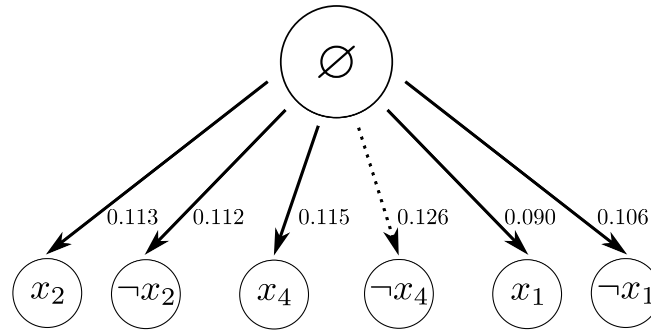


FIGURE 5.4 – Calcul des valeurs UCB dans UCTSAT.

Littéral	`passing`	`average`	Terme Exploration	UCB$_i$
x_2	10	0.108	0,005629155	0,113629155
$\neg x_2$	5	0.105	0,007960827	0,112960827
x_4	15	0.111	0,004596186	0,115596186
$\neg x_4$	7	0.120	0,006728127	0,126728127
x_1	10	0.085	0,005629155	0,090629155
$\neg x_1$	8	0.100	0,006293587	0,106293587

Nous remarquons que le terme exploration varie principalement en fonction de `passing` et de la variable $\mathcal{K}$: plus `passing` est élevé, plus le terme exploration devient petit. De même, plus le nombre d'itérations UCT est grand, plus ce même terme diminue, cela permet de tendre vers une exploration nulle petit à petit. Dans cette exemple, la variable $\mathcal{K}$ est trop petite, elle devrait s'ajuster dynamiquement suivant l'écart des moyennes. Une variable $\mathcal{K}$ mal réglée nous donne un terme d'exploration trop petit pour avoir un impact sur la recherche. Cette valeur devrait donc dans l'idéal s'adapter dynamiquement.

Création des sous-problèmes

L'étape de la descente peut nous renvoyer un nœud pour la sélection dans deux situations différentes. Rappelons que pour chaque nœud, nous descendons dans le fils ayant la plus grande valeur UCB associée, ou bien nous sélectionnons la valeur UCB spéciale. Cette valeur spéciale représente un autre dilemme exploration/exploitation. Lors de l'étape de la descente, nous retournons le nœud courant si la valeur UCB spéciale est plus grande que toutes les valeurs UCB des fils. Remarquons que même la racine peut calculer cette valeur UCB spéciale via `calculUCBspéciale()`. L'idée derrière cette fonctionnalité est de pouvoir diversifier les cubes. Nous pouvons à présent créer dans l'arbre un nouveau cube qui commence par un littéral différent : sans cette valeur UCB spéciale, cela n'est pas possible. D'une manière générale, les valeurs spéciales permettent d'avoir un arbre n-aire. En effet, cela nous permet de créer des fils sur un nœud qui en contient déjà. Puisque nous voulons beaucoup de cubes, qui se ressemblent le moins possible : le but est de ne pas explorer totalement l'arbre en profondeur. Un sous-cube est pour nous un cube qui est déjà contenu dans un ou plusieurs autres cube(s) plus grand(s).

Dans cette situation, le dilemme n'est plus le même. Devons nous ajouter des fils pour explorer, ou bien exploiter ceux déjà présents. Il est donc représenté mathématiquement d'une manière différente, l'indice UCB spécial associé au nœud i est :

$$\text{UCB}_i = \underbrace{\texttt{average}_i}_{\text{Terme Exploitation}} + \underbrace{\mathcal{K} * \sqrt{\frac{\ln(\texttt{passing}_i)}{\texttt{nbFils}}}}_{\text{Terme Exploration}}$$

Notons tout de même que la valeur UCB spéciale a dû être multipliée par une constante afin de l'associer aux autres valeurs UCB. De plus, retenons que pour un nœud, nous calculons lors de la descente, sa valeur UCB spéciale et les valeurs UCB de ses fils. Puis nous récupérons la valeur qui maximise toutes les autres. Si la valeur maximale appartient à un fils, nous descendons dans ce fils, sinon nous nous arrêtons sur ce nœud pour passer à l'étape de la sélection.

Conflits et littéraux déjà affectés

Algorithme 5.2 : `descenteUctSATV2` ($\mathcal{R}$)

Données : $\mathcal{R}$ la racine de l'arbre UCTSAT
Résultat : Le nœud utilisé pour l'étape de la sélection

```
1  Début
2    tant que possèdeFils(R) faire
3      si maxUCBSpécial(R) alors
4        retourner R;
5      SolveurSATdécide(R);
6      F ⟵ filMaxUCB(R);
7      si estConflit(F) ou déjàAffectéValeurOpposée(F) alors
8        supprime(F);
9        remontée(R);
10       return NULL;
11     si déjàAffectéMêmeValeur(F) alors
12       supprime(¬F);
13     passing(R) ⟵ passing(R) + 1;
14     R ⟵ F;
15   retourner R;
16 Fin
```

L'algorithme 5.2 met à jour la descente (algorithme 5.2) en ajoutant les explications précédentes ainsi que des nouveaux cas particuliers.

Au démarrage d'UCTSAT, le solveur associé est à la racine de son arbre de recherche et n'a pas encore pris de décisions. À chaque descente, le solveurs SAT associé décide les littéraux en question et effectue les propagations unitaires possibles. À ce moment, le solveur SAT peut réagir, nous avons ainsi trois cas différents :

- (Cas 1) le solveur renvoie une clause apprise et redémarre à cause d'un conflit sur un littéral ;
- (Cas 2) le littéral est déjà affecté, dû à la simplification des clauses apprises qui peuvent propager d'autres littéraux ;
- (Cas 3) tout se passe bien : le solveur a bien affecté un littéral.

Dans le cas d'un conflit (Cas 1), nous élaguons simplement le nœud en question (figure 5.5 et ligne 7 de l'algorithme), puis nous remontons les poids de l'arbre comme dans l'étape de remontée (nous

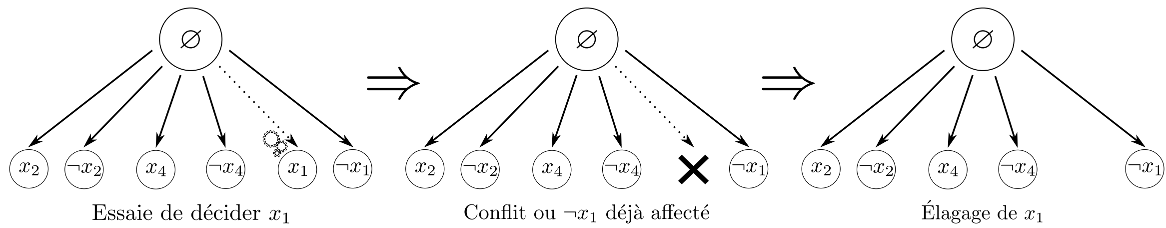


FIGURE 5.5 – Conflit ou littéral déjà affecté à une valeur opposée lors de la descente dans UCTSAT.

présentons cette étape dans la suite de ce chapitre, section 5.1.6). Par la suite, nous passons simplement à une nouvelle itération UCT.

Si un littéral est déjà affecté (Cas 2), nous avons deux situations possibles : soit ce littéral est affecté à une valeur égale à celle présente dans l'arbre (Cas 2-1), soit à son opposé (Cas 2-2).

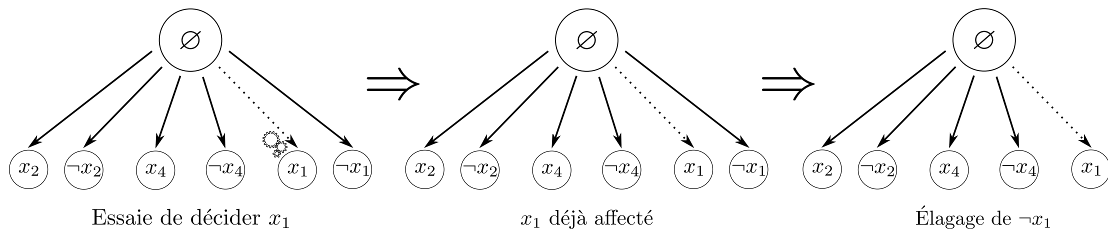


FIGURE 5.6 – Littéral déjà affecté lors de la descente dans UCTSAT.

Dans le premier cas (Cas 2-1), nous élaguons simplement le nœud représentant le littéral opposé dans le même niveau de l'arbre s'il est présent (figure 5.6 et ligne 11 de l'algorithme) et nous continuons le processus normal de l'algorithme. Dans le second cas (Cas 2-2), nous élaguons ce nœud comme pour un conflit (figure 5.5 et ligne 7 de l'algorithme), puis nous faisons remonter des poids (la remontée est expliquée dans la suite de ce chapitre, section 5.1.6) afin de mettre à jour les gains des nœuds parents. En traitant ainsi ces deux petits problèmes, nous assurons de ne pas retomber sur le même littéral (nœud) à la prochaine itération UCT. Rappelons que dans les cas où nous supprimons un nœud, nous redémarrons le solveur (*restart*, chapitre 2). Plus précisément, cela correspond à un redémarrage à la racine de son arbre pour le solveur SAT associé en gardant les clauses apprises. De plus, toujours dans cette situation, nous recommençons une itération UCT sans oublier de faire la remonté (section 5.1.6). Bien sûr, lorsque nous élaguons un tel nœud, nous supprimons récursivement tous ses descendants.

Une amélioration possible est de ne pas remonter le poids des nœuds. Cela nous permet de ne pas revenir à la racine de l'arbre et de continuer à choisir un autre littéral pour la descente. Nous accorderions alors à une itération UCT de faire son travail complètement afin de passer à l'étape de sélection. Cela permettrait de ne pas retourner au début d'une itération UCT à cause d'un conflit ou d'un littéral déjà affecté. De plus, le travail effectué par le solveur (application de la propagation unitaire) serait moins inutile car il ne redémarrerait pas. En contrepartie, la descente sera quelque peu faussée, car les poids n'auront pas été mis à jour.

5.1.4 Sélection

La figure 5.7 nous montre deux ajouts de nœuds différents : à gauche, nous avons un ajout de nœud sur une feuille de l'arbre tandis que sur la droite, nous avons un ajout nœud sur un nœud possédant déjà

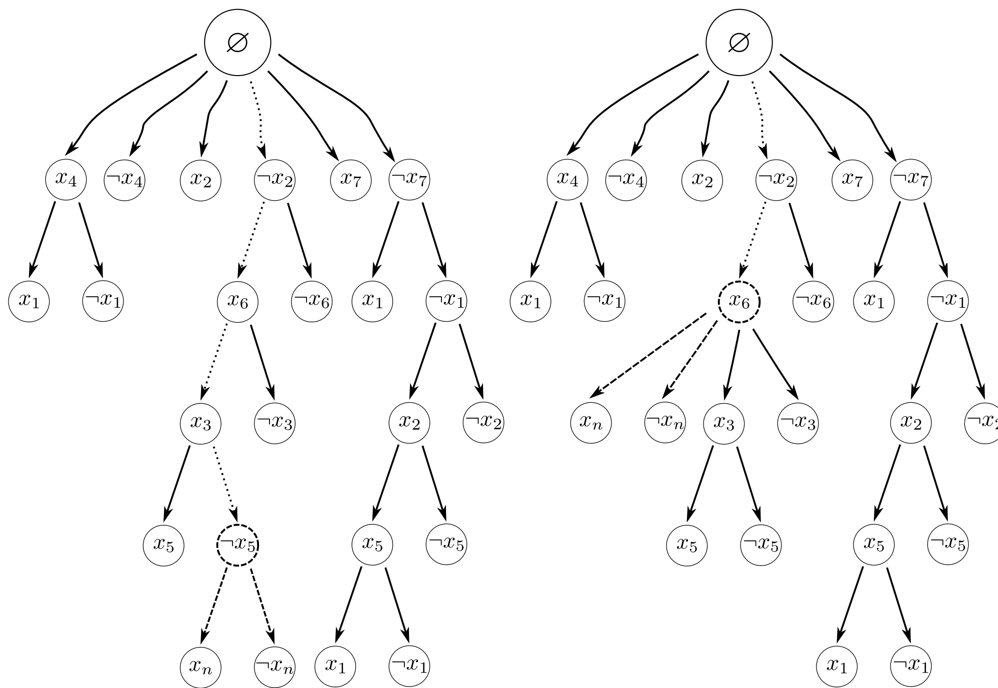


FIGURE 5.7 – La sélection dans UCTSAT.

des fils. Ce dernier cas est dû à la valeur UCB spéciale calculée durant la descente. Cette figure montre la sélection d'une variable après la descente (figure 5.3). Cette étape demande simplement au solveur SAT associé de lui fournir une nouvelle variable à ajouter dans l'arbre sur le nœud retourné lors de la descente. Cela est fait par l'heuristique de choix de variable d'un solveur SAT. Comme pour l'étape de descente, il se peut que l'un des deux littéraux de la variable choisie soit conflictuel, ou encore qu'il soit déjà affecté. Nous choisissons une variable qui n'appartient pas au fils du nœud en question pour ne pas avoir de répétition. Lors de cette sélection, nous créons deux nœuds pour la variable choisie : l'un représentant le littéral associé et l'autre, son complémentaire. Nous avons alors deux cas possibles pouvant subvenir lorsque nous souhaitons ajouter ces nouveaux nœuds :

- Seulement un de ces deux nœuds est problématique (conflit ou déjà affecté), dans ce cas, nous ajoutons à l'arbre uniquement le bon nœud (celui non conflictuel ou celui non affecté) ;
- Les deux nœuds posent problème : nous recommençons alors la sélection en choisissant une autre variable.

Nous avons besoin d'heuristiques à deux endroits distincts dans UCTSAT. La première heuristique est utilisée dans cette étape afin de choisir la prochaine variable à associer dans l'arbre de recherche UCT. C'est donc une heuristique de choix de variable comme nous pouvons en retrouver dans les solveurs SAT. La deuxième heuristique est en réalité la simulation d'un cube pour SAT. Remarquons que les heuristiques dans la littérature pour SAT peuvent servir directement à simuler un tel cube. Nous avons donc un large choix pour ces deux heuristiques.

Nous avons utilisé l'heuristique EVSIDS basée sur l'activité des conflits, pour choisir la prochaine variable à ajouter lors de l'étape de la sélection. Cette heuristique tend la recherche vers les conflits afin d'essayer de concentrer l'investigation vers la partie difficile du problème (voir section 2.3.3). Nous espérons ainsi récolter des cubes plus utiles. En contrepartie, nous obtenons plus de conflits. Cela a pour

effet d'élaguer plus encore l'arbre de recherche, et donc, de diminuer fortement le nombre de cubes récoltés à la fin. Le programme d'UCTSAT permet de prendre en considération différents solveurs SAT. Mais nous avons testé uniquement l'heuristique EVSIDS pour choisir les variables de l'arbre. Dans CUBEANDCONQUER, un choix de type *look-ahead* est utilisé : la variable choisie est celle produisant le plus de propagations unitaires (Heule *et al.* (2012), section 4.2.6).

5.1.5 Simulation

La simulation a pour but de donner le premier poids (*average*) au(x) nœud(s) venant d'être sélectionné(s). En effet, la sélection crée un ou deux nœuds suivant les conflits ou les littéraux déjà affectés. Une telle simulation va donc être faite sur le(s) cube(s) en question. Un cube est représenté par les littéraux sur le chemin de la racine de l'arbre à un nœud sélectionné. Pour cela, les heuristiques présentes dans la littérature pour SAT sont utiles car elles tentent d'estimer un cube pendant la recherche. Le solveur associé a décidé les littéraux en question lors de la descente, il est donc capable de faire cette simulation.

L'heuristique de simulation diffère par le fait qu'elle doit juger au mieux un cube. Une telle heuristique pourrait même être un solveur complet travaillant sur un tel cube. Cela coûterait un temps trop grand pour une méthode statique de création de cubes comme UCTSAT mais ceci pourrait être une idée dans une méthode parallèle future. Nous avons choisi comme simulation de prendre tout simplement le nombre de littéraux assignés au moment de la simulation d'un cube. C'est donc la même heuristique qu'utilise CUBEANDCONQUER pour choisir sa prochaine variable : le nombre de littéraux propagés. Cette simulation a l'avantage d'être simple et rapide, mais a pour inconvénient majeur de n'être pas assez précise sur la qualité d'un tel cube.

5.1.6 Remontée

Une fois la moyenne des nœuds de la simulation trouvée, il faut rééquilibrer par récurrence les ascendants associés. Pour cela, nous attribuons à un nœud, la moyenne de ses fils. Toutefois, notons que cette étape peut-être faite juste après la descente lors de la rencontre d'un conflit ou d'un littéral déjà affecté par le solveur SAT associé. La figure 5.8 représente la remontée dans l'arbre. Ces figures sont la suite directe des figures 5.3 (descente) et 5.7 (sélection). Grâce à la mise à jour des attributs *average* des nouveaux nœuds x_n et $\neg x_n$. Nous mettons à jour pour l'arbre représenté par la figure de gauche la moyenne (*average*), dans cet ordre, des nœuds représentant les littéraux $\neg x_5$, x_3, x_6 et $\neg x_2$. Sur la figure de droite (le cas ou un fil est crée dans un nœud possédant déjà des fils), nous avons respectivement la mise à jour des littéraux x_6 et $\neg x_2$. Cela permet de rectifier le déséquilibre dans les moyennes des nœuds de l'arbre apporté par les deux nouveaux nœuds créés lors de la sélection : x_n et $\neg x_n$.

5.1.7 Récupération des cubes

Une fois le bon nombre d'itérations UCT effectué, nous trions les cubes via leur moyenne dans l'arbre (*average*). L'algorithme 5.3 illustre la méthode utilisée, elle se contente de prendre le cube contenant la meilleure moyenne (ligne 4 à 6) puis d'effacer la feuille en question. Quand nous effaçons cette feuille, nous effaçons aussi récursivement ses ascendants quand ils deviennent des feuilles (ligne 8 à 11). Nous réitérons ce procédé afin de récupérer les cubes par rapport à leur moyenne. Nous obtenons donc un ordre sur les interprétations partielles.

Nous allons maintenant exposer une autre approche de décomposition statique.

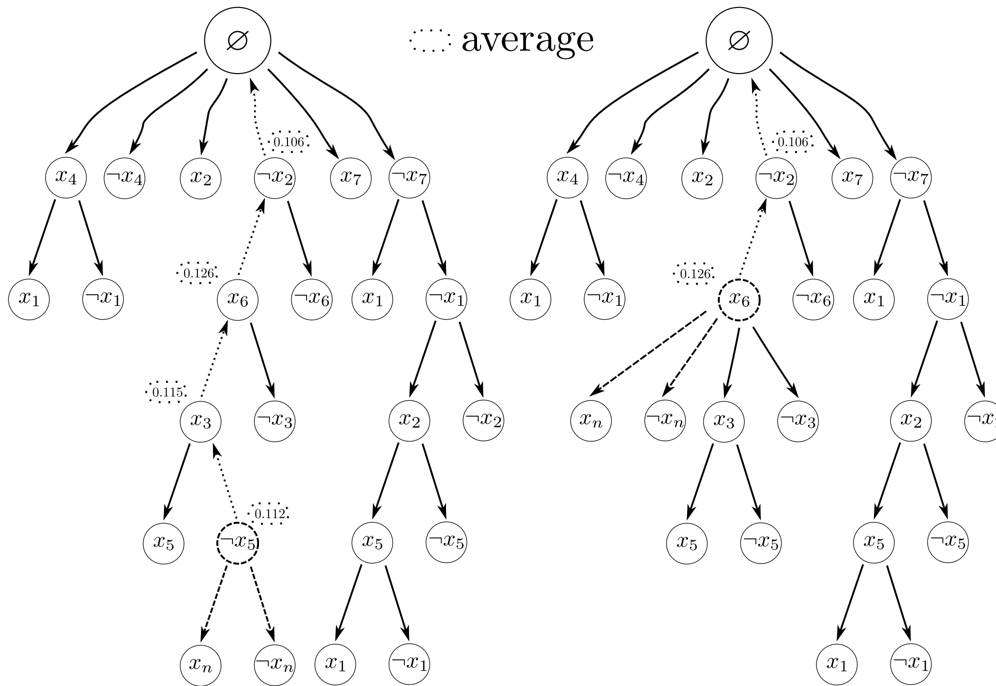


FIGURE 5.8 – La remontée dans UCTSAT.

5.2 Décomposition via CUBEANDCONQUER : SWARMSAT

Dans cette section, nous décrivons le solveur SWARMSAT qui a donné lieu à la publication nationale Audemard *et al.* (2015). Ce solveur essaye de résoudre les cubes (sous-problèmes) crées par CUBEAND-CONQUER (section 4.2.6, Heule *et al.* (2012)). Nous présentons trois types de processus utilisés dans SWARMSAT, ainsi que la manière dont les communications entre ces derniers sont réalisées.

5.2.1 Les différents processus

Le scout : Il se charge de la construction des cubes de la même manière que CUBEANDCONQUER (ie. utilisation du solveur MARCH_CC (Heule et van Maaren 2006b)).

Le worker : C'est un solveur SAT qui se charge de la résolution des cubes. Afin de profiter du travail déjà réalisé et d'obtenir une raison lors de l'échec de la résolution de ce dernier, le cube sera passé en *assumption* et le problème résolu de manière incrémentale (voir (Audemard *et al.* 2014a) pour plus d'informations). Ainsi, le résultat retourné par le solveur peut être de deux types : soit il répond SAT et le problème est donc montré globalement satisfaisable ; soit le cube est réfuté. Dans ce dernier cas, il est possible, grâce à la notion d'*assumption*, d'extraire une raison de l'insatisfaisabilité du cube. Si la clause ainsi générée est vide, l'insatisfaisabilité est donc montré globalement, sinon l'insatisfaisabilité est locale et le *worker* demande un nouveau cube à résoudre.

Le master : Il se charge de l'initialisation et de la communication entre les *workers* et le *scout*. C'est ce dernier qui va s'occuper de gérer les cubes produits par le *scout* (ils sont stockés dans une file) et qui les distribue au *worker* en attente. Le *master* a aussi la tâche de centraliser les clauses provenant des cubes déjà prouvés insatisfaisables. Grâce à ces dernières, il a également la possibilité de ne pas considérer un cube qui contredit une des clauses.

Algorithme 5.3 : `récupCube` ($\mathcal{R}$)

Données : $\mathcal{R}$ la racine de l'arbre
Résultat : $\mathcal{C}$ l'ensemble ordonné des cubes récupérés

```
1  Début
2    tant que possèdeFils(R) faire
3      N ⟵ R;
4      tant que possèdeFils(N) faire
5        N ⟵ filMaxUCT(N);
6      C ⟵ C ∪ récupCubeFeuille(N);
7      P ⟵ père(N);
8      tant que possèdeUnSeulFils(P) faire
9        delete(N);
10       N ⟵ P;
11       P ⟵ père(P);
12   retourner C;
13 Fin
```

5.2.2 Communication

Les communications sont réalisées point-à-point via des messages non-bloquants. Elles sont utilisées pour la transmission des cubes ainsi que le passage de certaines clauses des *workers* au *master* ou encore pour la transmission de la solution. Remarquons que dans la version actuelle de SWARMSAT, aucune des clauses apprises n'est échangée entre les *workers*.

5.2.3 Mode de fonctionnement

SWARMSAT est un solveur modulaire qui permet d'ajouter facilement de nouveaux types de *workers*, de gérer une création de cubes dynamique ou statique et aussi de résoudre une instance du problème SAT en parallèle que ce soit en mode collaboratif (à la CUBEANDCONQUER) ou concurrentiel (à la PPFOLIO (Roussel 2011)).

La gestion de l'ajout d'un nouveau solveur est ainsi réalisée grâce à l' implémentation d'une interface écrite en C++. Elle permet de définir les méthodes nécessaires à la gestion des communications via OPEN MPI. Dans la version expérimentée de ce papier, nous avons considéré les solveurs GLUCOSE (Audemard et Simon 2012) et MINISAT (Eén et Sörenson 2004) pour implémenter les *workers*.

En ce qui concerne le scout, nous utilisons la version de MARCH_CC fournie par les auteurs de CUBEANDCONQUER. Cependant, nous avons aussi proposé une version dynamique qui, contrairement à la version initiale, permet de rendre disponible les cubes durant leur génération (dans CUBEANDCONQUER il fallait parfois attendre plus de 20 minutes pour obtenir tous les cubes). Cela nous permet d'anticiper la résolution des cubes même si en contrepartie, certains cubes auraient été prouvés insatisfaisables durant la phase de création.

Dans la suite, nous nommerons SWARMSAT($\texttt{W}$, $\texttt{S}$, $\texttt{G}$)$_{\texttt{C}}$ pour signifier que nous utilisons comme *worker* des solveurs de type `W` (GLUCOSE, MINISAT, ...) et un *scout* utilisant la méthode `S` pour la génération des cubes. La lettre `G` peut prendre comme valeur `dynamic` ou `static` en fonction de la disponibilité des cubes (directement ou après leur création) tandis que `C` permet de connaître le nombre de workers utilisés.

5.3 Expérimentations

5.3.1 UCTSAT

Pour ces expérimentations, nous avons sélectionné 7 instances de la compétition SAT 2013. Puis nous avons testé les 100 premiers cube puis 100 autres cubes répartis sur le reste des cubes afin d'essayer d'avoir une vision globale de la difficulté des cubes. Nous avons exécuté UCTSAT avec 5000 itérations UCT, puis sur 20000 itérations UCT. Nous discuterons uniquement des tests effectués sur 5000 itérations UCT sachant que ceux sur 20000, nous apportent les mêmes résultats. Quand il y a un tiret, c'est que l'on à aucun cube satisfaisable. Notons que le temps accordé à la création des cubes est de 2 heures.

Instance	Clauses	Variables	Cubes	%sous-cubes	UCB	Conflits	Fils spéciaux
003.log	72259	4128	5085	47	Explore souvent	41	116
aes641keyfind1	2632	596	4918	10	Bien	70	22
bob12m06	1369224	482635	8512	8	Fait qu'explorer	75	3644
pb30001lb00	595953	140866	4297	0.16	Bien	412	5
pb30002lb06	630463	148565	4936	1.05	Bien	54	19
pb20003lb03	311884	77289	1348	0	Bien	2088	27
UTI205p1	1195016	225926	4933	0.16	Bien	70	7

TABLE 5.1 – Information sur la création des cubes par UCTSAT.

La tableau 5.1 expose des informations liées à la création des cubes par UCTSAT. Le pourcentage de sous-cubes est le pourcentage du nombre de cubes étant subsumés. Pour rappel, un sous-cube est pour nous un cube qui est inclus dans un ou plusieurs autres cubes plus grands. À part pour une instance, ce pourcentage est assez bas. Cela montre que les sous-problèmes sont assez diversifiés car les cubes associés sont alors différents. Remarquons que sur les deux instances où l'UCB est mal réglée (cellules « Explore souvent » et « Fait qu'explorer »), nous avons un pourcentage plus élevé de sous-cubes. Ce défaut pourrait devenir problématique si nous utilisons d'autres heuristiques. Une solution consisterait à régler dynamiquement les valeurs UCB. Le nombre de conflits est assez faible à cause de l'exploration qui est faite, de sorte que, la recherche est moins conflictuelle.

Instance	NbCube SAT	NbCube UNSAT	NbCube TIMEOUT
003	117	0	83
aes641keyfind1	0	196	4
bob12m06	0	183	17
pb30001lb00	21	37	142
pb30002lb06	1	139	60
pb20003lb03	11	188	1
UTI205p1	0	200	0

TABLE 5.2 – Résultats de UCTSAT sur 7 instances.

Le tableau 5.2 expose les résultats concernant la satisfaisabilité des cubes. Nous avons donc quatre instances sur les sept qui nous ont donné des cubes satisfiables parmi les 200 premiers cubes. Ces résultats montrent que nous obtenons quelques instances satisfaisables. En revanche, seul une instance insatisfaisable sur les trois a été prouvées (UTI205p1).

5.3.2 SWARMSAT

L'ensemble des expérimentations a été réalisé sur 64 cœurs grâce à deux nœuds de 32 cœurs. Les nœuds de calcul sont des Dell R910 avec 4 Intel Xeon X7550 contenant chacun huit cœurs. Chaque unité de calcul dispose d'un contrôleur Gigabit Ethernet et de 256 Go de RAM.

Les expérimentations présentées dans ce qui suit utilisent le même protocole (instances et temps de résolution) que celui proposé par les auteurs de DOLIUS (Audemard *et al.* 2014c). Plus précisément, nous considérons 20 minutes comme limite de temps réel. Les instances sélectionnées sont issues de la compétition SAT 2013 (*application track*) (Balint *et al.* 2013). Une instance est sélectionnée si elle a été résolue par au moins un des cinq premiers solveurs parallèles de la compétition et ne peut être résolue par tous ces solveurs. Cela permet d'avoir des instances difficiles et diversifiées : 18 satisfaisables et 33 insatisfaisables.

Dans SWARMSAT, étant donné la nature du *scout* (durée limitée) et du *master* (peu de travail), nous avons choisi de faire cohabiter ces derniers avec les *workers*.

Nous comparons le solveur DOLIUS (Audemard *et al.* 2014c) avec différentes versions de SWARMSAT présentées subséquemment :

- $S(G,\emptyset,\emptyset)_{64}$ pour SWARMSAT(GLUCOSE $_{rdm}$, $\emptyset$, $\emptyset)_{64}$: la génération des cubes étant vide, cette version est un solveur concurrentiel. Afin que les solveurs rentrent en concurrence nous utilisons une version de GLUCOSE avec une graine aléatoire différente pour chaque *worker* ;
- $S(G,m,d)_{64}$ pour SWARMSAT(GLUCOSE, MARCH_CC, dynamic$)_{64}$: nous utilisons 64 *workers* de type GLUCOSE et MARCH_CC en mode dynamique pour la génération des cubes ;
- $S(M,m,d)_{64}$ pour SWARMSAT(MINISAT, MARCH_CC, dynamic$)_{64}$: nous utilisons 64 *workers* de type MINISAT et MARCH_CC en mode dynamique pour la génération des cubes ;
- $S(G,m,s)_{64}$ pour SWARMSAT(GLUCOSE, MARCH_CC, static$)_{64}$: nous utilisons comme *workers* 64 solveurs GLUCOSE et MARCH_CC en mode statique. Ici, les *workers* tentent de résoudre l'instance en deux étapes. Dans un premier temps, sans cube durant le temps nécessaire à l'exécution du *scout*. Puis dans un second temps, via les cubes.

Solveur	SAT (18)	UNSAT (33)	Total (51)
$S(G,\emptyset,\emptyset)_{64}$	12	12	24
$S(G,m,s)_{64}$	10	11	21
$S(M,m,d)_{64}$	10	1	11
$S(G,m,d)_{64}$	6	4	10
DOLIUS	**13**	**24**	**37**

TABLE 5.3 – Résultats des différentes versions de SWARMSAT contre DOLIUS.

Le tableau 5.3 reporte les résultats obtenus par les différentes approches expérimentées dans ce papier. Tout d'abord, nous discutons des résultats obtenus par le solveur DOLIUS. Nous pouvons remarquer que ce dernier est significativement meilleur sur les instances insatisfaisables (24 instances insatisfaisables résolues) que les différentes versions de SWARMSAT testées (12 pour la meilleure version de SWARMSAT). Cela s'explique largement par le fait que contrairement à SWARMSAT, DOLIUS permet d'échanger les clauses apprises entre les différentes unités de calcul. Cet échange d'informations permet de guider les solveurs plus rapidement vers la génération d'une clause vide. Pour ce qui est des instances satisfaisables, nous pouvons voir que l'impact de l'échange d'informations n'est pas aussi prépondérant et qu'une version concurrentielle simple telle que $S(G,\emptyset,\emptyset)_{64}$ permet de résoudre approximativement le même nombre d'instances (13 pour DOLIUS et 12 pour $S(G,\emptyset,\emptyset)_{64}$).

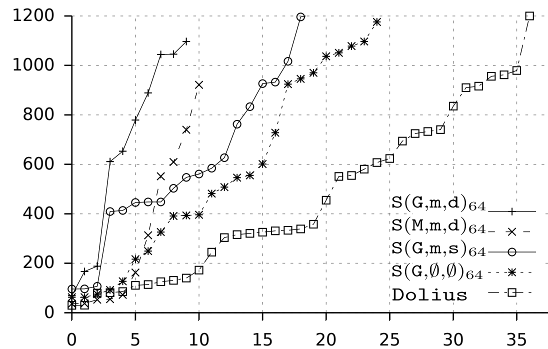


FIGURE 5.9 – Résultats des différentes versions de SWARMSAT contre DOLIUS (*cactus plot*).

Concernant les résultats obtenus par les différentes versions de SWARMSAT, nous observons qu'une division de l'espace de recherche à l'aide de cubes, telle qu'elle a été proposée dans (Heule *et al.* 2012), n'est pas efficace si elle n'est pas accompagnée d'un échange d'informations entre les solveurs. En effet, la version concurrentielle de SWARMSAT $\texttt{S(G,}\emptyset,\emptyset)_{64}$, qui ne divise pas l'espace de recherche à l'aide de cubes, fournie les meilleurs résultats (12 instances satisfaisables résolues et 12 instances insatisfaisables résolues). De plus, comme le montre la figure 5.9, $\texttt{S(G,}\emptyset,\emptyset)_{64}$ résout aussi plus rapidement les instances. Afin de vérifier que la perte de performances de l'approche basée sur les cubes n'était pas due au coût de communication, nous avons mesuré ces derniers. Il s'avère que le temps de transfert des cubes du *master* aux *workers* ainsi que le passage des clauses apprises des *workers* au *master* est insignifiant (moins de 2 secondes au total).

Les expérimentations ne permettent pas de conclure sur la qualité des cubes. En effet, bien que $\texttt{S(G,m,s)}_{64}$ résout significativement plus d'instances (10 instances satisfaisables et 11 instances insatisfaisables) que $\texttt{S(G,m,d)}_{64}$ (6 instances satisfaisables et 4 instances insatisfaisables) et $\texttt{S(M,m,d)}_{64}$ (10 instances satisfiables et 1 instance insatisfaisable), dans la plupart des cas cela est une conséquence du fait que la génération de tous les cubes est trop coûteuse en temps. Pour 33 instances la génération ne termine pas et pour les 18 instances restantes, cette dernière nécessite en moyenne 296.27 secondes du temps alloué à la résolution. Ces chiffres montrent que dans la majeure partie du temps $\texttt{S(G,m,s)}_{64}$ est équivalent à $\texttt{S(G,}\emptyset,\emptyset)_{64}$. Cela explique largement les performances de $\texttt{S(G,m,s)}_{64}$.

Finalement, nous pouvons voir que le choix du solveur utilisé pour implémenter les *workers* est prépondérant. Ainsi, le choix du solveur MINISAT permet de résoudre plus facilement les instances satisfaisables tandis qu'utiliser GLUCOSE permet d'être plus efficace sur les instances insatisfaisables. Ces résultats s'expliquent par le fait que cette différence de performance en fonction de la satisfaisabilité de l'instance existe déjà entre ces deux solveurs lorsqu'ils sont exécutés séquentiellement. Néanmoins, il semble que dans le cas d'une résolution incrémentale le phénomène est accentué.

5.4 Conclusion

Nous avons exposé deux méthodes de construction de cubes dans une approche « diviser pour mieux régner » dans le cadre de SAT.

La première méthode a l'originalité de s'attaquer pour la première fois à ce problème en utilisant l'algorithme UCT, qui tente de trouver un bon compromis entre l'exploitation et l'exploration. De plus, cette méthode a été faite dans le but de créer un futur solveur massivement parallèle. La méthode UCTSAT respecte donc très bien son rôle. Cependant, beaucoup d'améliorations doivent encore être réalisées dans UCTSAT, notamment au niveau des heuristiques choisies. En effet, il est préférable de tester plusieurs

heuristiques pour ainsi sélectionner la meilleure : cela est l'objectif de travaux futurs. Une amélioration possible est de transformer cette méthode en utilisant la puissance de la parallélisation, afin d'affiner les simulations faites dans UCTSAT.

La deuxième méthode présente un nouveau solveur SAT, nommé SWARMSAT, largement inspiré de CUBEANDCONQUER et dédié au massivement parallèle. Cette première mouture utilise pour la génération des cubes le solveur MARCH_CC (Heule *et al.* 2012) et comme *workers* l'un des deux solveurs de l'état de l'art GLUCOSE et MINISAT. Les résultats expérimentaux montrent que dans le cadre de la résolution d'instances industrielles SWARMSAT (plus généralement, les approches fondées sur CUBEANDCONQUER) ne sont pas efficaces en pratique (sauf dans le cas où les instances seraient de type académique).

Bien que les résultats obtenus ne soient pas à la hauteur de nos attentes, les travaux réalisés autour de la création des solveurs UCTSAT et SWARMSAT forment une base solide pour de nombreuses extensions. En effet, nous avons la possibilité d'actionner plusieurs leviers afin d'améliorer ces approches.

Un fait marquant dans les expérimentations de SWARMSAT est que pour beaucoup d'instances le nombre de cubes résolus est insignifiant (par exemple pour `bob12s06` seulement 4% des cubes ont été traités). Une telle situation peut conduire à focaliser la recherche sur une seule partie de l'espace de recherche. Cela peut être très préjudiciable lorsque nous observons l'impact des redémarrages dans les solveurs SAT modernes.

Afin d'atténuer ce phénomène, nous avons réalisé des travaux dans le prochain chapitre (AMPHAROS, le chapitre 6). C'est travaux font faire en sorte que des cubes puissent obtenir le statut indéterminé. De cette manière un *worker* peut libérer un cube et en choisir un autre. Par la suite, le cube ainsi libéré pourra être retravaillé. Les travaux liés à AMPHAROS améliore l'échange d'informations entre les *workers*. Il est tout à fait envisageable de partager des clauses entre les différentes unités de calcul. Cependant, une attention particulière devra être apportée à la manière dont cet échange sera réalisé. Pour cela, nous avons étudié plusieurs modèles de programmation distribués via une contribution exposée dans le chapitre 7 (D-SYRUP).

AMPHAROS : Un solveur « diviser pour mieux régner » distribué

Chapitre **6**

Sommaire

6.1 La gestion de l'arbre 136
6.1.1 Initialisation 137
6.1.2 Transmission 138
6.1.3 L'extension 139
6.1.4 L'élagage 140
6.2 L'échange des clauses 141
6.2.1 Le partage classique des clauses apprises 141
6.2.2 Les littéraux unitaires sous hypothèses 142
6.3 Intensification vs Diversification 144
6.3.1 Évaluation du degré de redondance 144
6.4 Évaluation expérimentale 146
6.4.1 Gestion des communications 146
6.4.2 Configuration 146
6.4.3 Résultats 147
6.5 Conclusion 151
6.5.1 MAPLEAMPHAROS : Le ratio de propagations par décision 153

Nous présentons dans ce chapitre un solveur distribué nommé AMPHAROS. Ce solveur basé sur un paradigme « diviser pour mieux régner », à l'originalité de partager des informations (en plus des clauses apprises) qui sont dépendantes de sous-problèmes générés dynamiquement durant la recherche. De ces échanges, d'autres informations peuvent être déduites et transmises aux autres solveurs. Ces travaux ont donnés lieu à deux publications, une internationale Audemard *et al.* (2016a) et l'autre francophone Audemard *et al.* (2016b).

Nous présentons d'abord d'une manière générale AMPHAROS (section 6.1), ce solveur met en place un arbre de guidage comme dans MTSS (section 4.2.2, page 106, Vander-Swalmen *et al.* (2008; 2009)). Notons cependant, que cet arbre n'est pas partagé entre tous les *threads*. Plus précisément, seul un *thread* le possède et celui-ci envoi quelques chemins de guidage incluent dans l'arbre de guidage à certains *threads*. AMPHAROS met en œuvre des fonctionnalités qui mettent en avant une nouvelle approche « diviser pour mieux régner ». Cette approche a notamment la particularité d'ajuster dynamiquement la division en fonction des informations échangées entre les solveurs et de pouvoir placer plusieurs solveurs sur le même sous-problème d'une manière concurrentielle. Ces fonctionnalités répondent à plusieurs questions très importantes dans une telle approche :

- Comment démarrer la recherche ? (section 6.1.1) ;
- Comment les solveurs choisissent leurs sous-problèmes à résoudre ? (section 6.1.2) ;
- Comment diviser le problème en sous-problèmes ? (section 6.1.3) ;
- Comment l'arbre binaire (de guidage) est résolu ? (section 6.1.4) ;

- Quelles informations pouvons nous partager entre les solveurs ? (section 6.2) ;
- Pouvons-nous diviser la recherche en fonction des informations partagées ? (section 6.3).

Pour finir, avant de conclure (section 6.5), nous étudions expérimentalement AMPHAROS. Plus précisément, nous démontrons l'utilité de ses composants (section 6.4.3), nous montrons sa bonne extensibilité (section 6.4.3), nous le comparons contre quelques solveurs de l'état de l'art (section 6.4.3) puis nous démontrons expérimentalement l'utilité d'un ratio permettant de diviser le problème en fonction des informations échangées (section 6.4.3).

6.1 La gestion de l'arbre

Les performances des approches « diviser pour mieux régner » dépendent surtout de l'efficacité de la division de l'espace de recherche mais aussi de l'assignation des solveurs aux sous-espaces. Même si AMPHAROS est un solveur de type « diviser pour mieux régner », il est important de noter que, contrairement à (Schubert *et al.* 2009) ou PSATO (section 4.2.1, page 104, Zhang *et al.* (1996)), il n'utilise pas la stratégie de vol de travail (*work stealing*) afin de donner des sous-problèmes aux solveurs. Dans notre cas, la division est faite d'une manière classique comme dans (Audemard *et al.* 2014b, Chu *et al.* 2008). Plus précisément, notre approche génère des chemins de guidages, réduit aux cubes et couvrant tout l'espace de recherche. De cette manière, le résultat de cette division est un arbre où les nœuds sont des variables et les branches gauches (resp. droites) correspondent à l'assignement de la variable (du nœud de départ de la branche) à vrai (resp. faux). Les solveurs opèrent sur les feuilles (représentées par le symbole NIL) et résolvent (sous hypothèses) la formule initiale restreinte à un cube correspondant au chemin de la racine à la feuille associée. La figure 6.1 montre l'exemple d'un tel arbre contenant trois feuilles (les cubes $[x_1, \neg x_2, x_4]$, $[x_1, \neg x_2, \neg x_4]$ et $[\neg x_1, \neg x_3]$), deux branches fermées (déjà prouvées insatisfaisables) et trois solveurs ($S_1 \dots S_3$) travaillant sur ces feuilles.

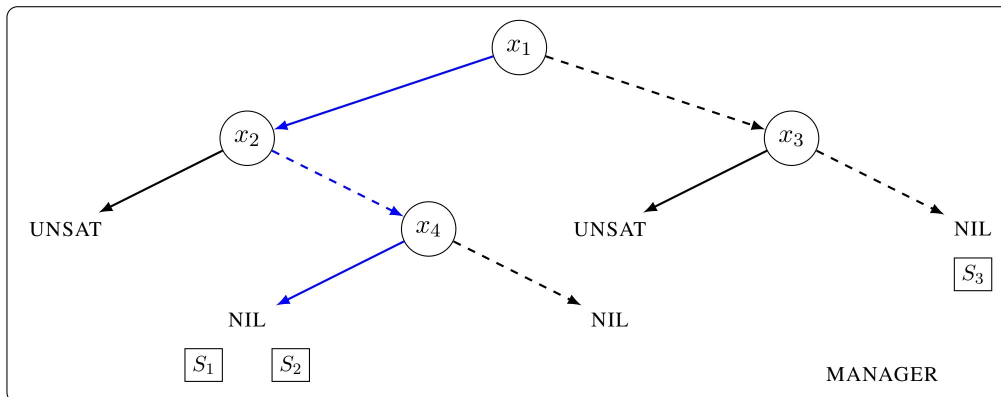


FIGURE 6.1 – Architecture globale d'AMPHAROS.

Comme nous allons le voir dans la section 6.1.2, dans notre architecture, les solveurs peuvent travailler sur le même cube (comme les solveurs S_1 et S_2 dans la figure 6.1) et s'arrêter avant de trouver une solution ou une contradiction. Dans AMPHAROS, à chaque fois qu'un solveur partage des informations ou demande un nouveau cube à résoudre, il communique avec un processus dédié appelé le MANAGER. Sa principale mission est de gérer l'arbre de division (les cubes) et la communication entre les solveurs (de type CDCL). Ainsi, quand un solveur prend la décision d'arrêter de résoudre un cube donné (sans l'avoir résolu), il peut demander au MANAGER de l'étendre (voir Sect. 6.1.3). Un solveur peut aussi arrêter de résoudre un cube donné quand il le prouve insatisfaisable et dans ce cas, un message informe

le MANAGER afin de mettre à jour l'arbre en conséquence (voir section 6.1.4). Dans ces deux cas, à chaque fois qu'un solveur stoppe la résolution d'un cube, il va communiquer avec le MANAGER afin de descendre dans l'arbre et choisir lui-même un nouveau cube à résoudre (potentiellement le même, voir section 6.1.2). La fin du processus de résolution arrive finalement quand un cube est prouvé satisfaisable ou quand l'arbre complet est prouvé insatisfaisable.

6.1.1 Initialisation

Au départ du processus de recherche, nous initialisons les *workers*. Cette étape permet d'initialiser l'activité des variables (associées à l'heuristique VSIDS (Moskewicz *et al.* 2001b)) ainsi que leurs polarités et de créer la racine de l'arbre. Pour cela, tous les solveurs essaient de résoudre en concurrence la formule complète pendant un certain nombre de conflits ($10,000$ dans notre implémentation). Cela correspond à résoudre la formule sans hypothèse (avec un cube vide). Afin de diversifier la recherche, la première descente de chaque solveur est faite aléatoirement (*i.e.* le choix des variables et leurs polarités). Puis, de la même manière que dans (Martins *et al.* 2010), le premier solveur atteignant le nombre maximum de conflits communique sa meilleure variable suivant l'heuristique VSIDS au MANAGER. La figure 6.2 présente cette initialisation avec n solveurs.

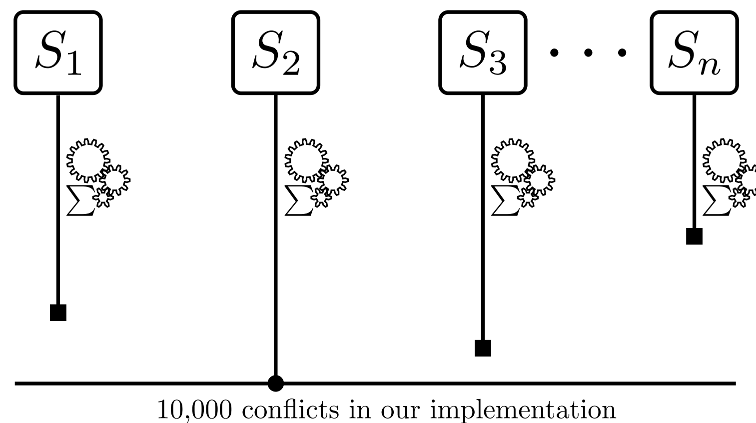


FIGURE 6.2 – Initialisation d'AMPHAROS : le premier solveur atteignant le nombre maximum de conflits communique sa meilleure variable suivant l'heuristique VSIDS au MANAGER.

Cette variable devient la racine de l'arbre et tous les solveurs arrêtent leurs recherches concurrentielles afin de demander un cube au MANAGER. Initialement, l'arbre contient donc seulement deux feuilles, *i.e.* les cubes sont restreints à un seul littéral (la variable et son opposé). Dans la figure 6.3, la variable sélectionnée est x_1 et l'ensemble initial des cubes est $\{[x_1], [\neg x_1]\}$.

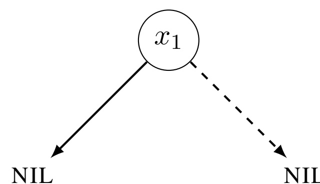
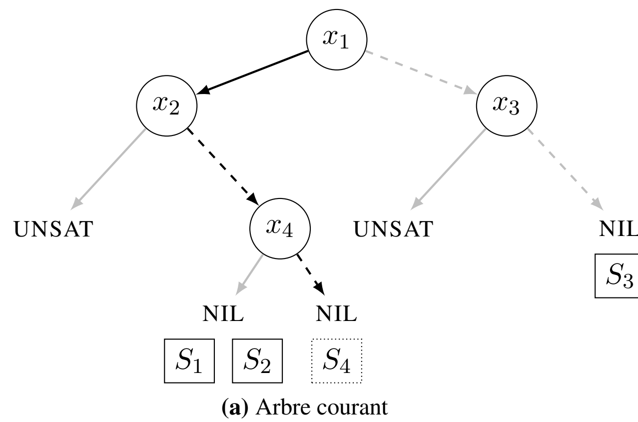


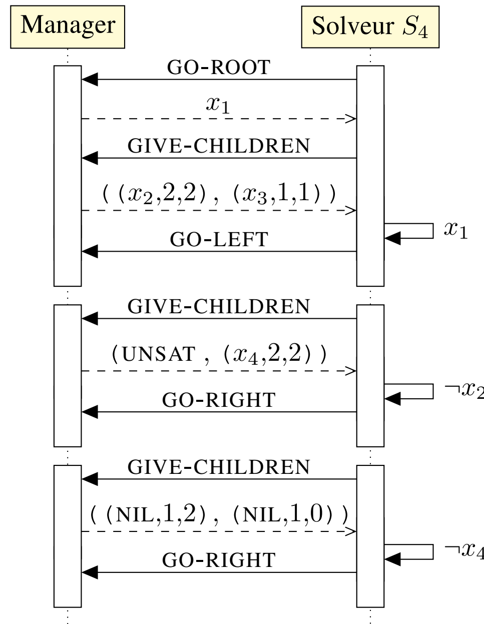
FIGURE 6.3 – Initialisation de l'arbre d'AMPHAROS : la variable sélectionnée est x_1 et l'ensemble initial des cubes est $\{[x_1], [\neg x_1]\}$.

6.1.2 Transmission

Comme indiqué au préalable, un solveur peut arrêter la recherche avant d'avoir fini de résoudre un cube. Cette situation se produit quand il n'arrive pas à résoudre le sous-problème associé au cube avant un certain nombre de conflits (fixé à 10000 ici). Le solveur contacte alors le MANAGER afin de sélectionner un nouveau cube à résoudre (qui peut être le même). L'originalité de notre méthode est de laisser le solveur choisir son cube lui-même parmi tous ceux non résolus dans l'arbre (correspondant aux feuilles NIL). La Fig. 6.4 montre un diagramme de séquence (Fig. 6.4b) qui illustre les messages échangés lorsque le solveur S_4 demande un nouveau cube au MANAGER (Fig. 6.4a).



(a) Arbre courant



(b) Diagramme de sequence

FIGURE 6.4 – Vue d'ensemble des messages échangés entre le solveurs S_4 et le MANAGER (Fig 6.4b) suivant une division de cubes sous forme d'arbre (Fig 6.4a) (une ligne pleine (resp. en pointillés) affectes la variable à vrai (resp. faux)).

Un premier message (GO-ROOT) est envoyé par le solveur pour demander la variable associée à la racine de l'arbre. Il reçoit x_1. Par la suite, à chaque étape de sélection d'une branche, le solveur demande les variables des nœuds fils de la variable précédemment reçue (message GIVE-CHILDREN).

La réponse est composée de deux triplets : une pour chaque polarité du nœud courant. Chaque triplet est composé, dans cet ordre, de la variable du fils, du nombre de feuilles disponibles (nœuds NIL) et du nombre de solveurs travaillant à cet instant sur ces feuilles. Dans la Fig. 6.4b, la réponse du MANAGER au premier message GIVE-CHILDREN est le triplet $(x_2, 2, 2)$ pour la polarité positive de x_1 (la branche gauche contient deux feuilles, et deux solveurs travaillent sur ces feuilles (S_1 et S_2)) et le triplet $(x_3, 1, 1)$ pour la négative.

À ce moment, le solveur va prendre une décision selon les valeurs des triplets : soit il va descendre vers la gauche (affecter positivement la variable courante) soit vers la droite (affecter négativement la variable courante). Par défaut, il choisit la branche possédant un nombre de solveurs inférieur au nombre de feuilles. L'idée est de couvrir le plus de cubes dans l'arbre et ainsi diversifier la position des solveurs. Si pour les deux branches, nous avons la même réponse (vrai-vrai et faux-faux), alors le solveur sélectionne la branche en fonction de son vecteur de polarité (sa *phase-saving*) (Pipatsrisawat et Darwiche 2007). Après avoir sélectionné sa branche, le solveur informe le MANAGER (via le message GO-LEFT ou GO-RIGHT) et affecte le littéral associé comme hypothèse.

Ainsi, dans la Fig. 6.4, le solveur S_4 affecte x_1 (la racine) positivement (la condition mentionnée précédemment est fausse pour les deux branches). Par la suite, étant donné qu'une des branches liée à x_2 est déjà prouvée insatisfaisable, S_4 n'a pas d'autre alternative que d'affecter x_2 à faux (négativement). Puis, il doit mettre x_4 à faux puisque la condition précédente est prise en compte. Enfin, le solveur est arrivé sur une feuille (NIL) et peut commencer à résoudre le cube $[x_1, \neg x_2, \neg, x_4]$.

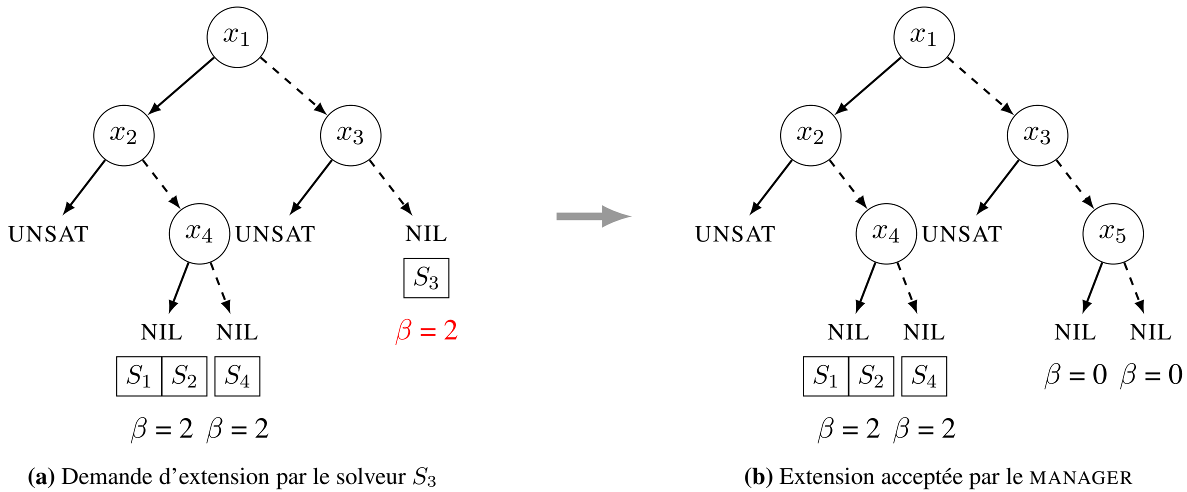
6.1.3 L'extension

Initialement, l'arbre contient une seule variable et deux cubes à résoudre (voir Sect. 6.1.1). Pour diviser la formule originale en un nombre de cubes conséquent, nous proposons d'étendre dynamiquement l'arbre pendant la recherche. Rappelons que nous n'utilisons pas une stratégie de type *work stealing*.

Nous associons à chaque feuille de l'arbre une variable entière β représentant la difficulté supposée du sous-problème associé. À chaque fois qu'un solveur arrête de résoudre un cube (sans trouver de solution), la variable β de la feuille associée à ce cube est incrémentée. De ce fait, plus β est grand, plus le cube associé est potentiellement difficile à résoudre (car beaucoup de solveurs n'ont pas réussi à le résoudre). Notons qu'un même solveur peut incrémenter plusieurs fois la même variable β. À chaque fois qu'un solveur demande un nouveau cube, la valeur β de la feuille associée est examinée. Si elle est plus grande ou égale au nombre de feuilles disponibles (*i.e.* les feuilles NIL) fois un facteur d'extension f_e, alors l'arbre est étendu au niveau de la feuille associée.

L'extension est faite de cette manière : le dernier solveur incrémentant la variable β transmet au MANAGER sa meilleure variable booléenne suivant l'heuristique VSIDS puis deux nouvelles feuilles sont créées afin d'étendre le cube associé. La valeur β des deux nouvelles feuilles est initialisée à 0. Le fait de prendre en compte le nombre de feuilles disponibles permet de gérer le nombre de cubes : plus l'arbre contient de cubes, moins nous aurons d'extensions. De cette manière et contrairement à Cube And Conquer (van der Tak *et al.* 2014), notre approche évite de créer un trop grand nombre de cubes, en prenant compte des cubes déjà prouvés insatisfaisables. Comme une feuille peut contenir plusieurs solveurs, notons qu'après une extension, quelques solveurs peuvent travailler sur des nœuds qui ne sont pas des feuilles.

La Fig. 6.5 montre l'exemple d'une extension. L'arbre (le même que la Fig. 6.4) contient 3 feuilles disponibles et quelques solveurs travaillent sur ces feuilles. Quand le solveur S_3 arrête de résoudre le cube $[\neg x_1, \neg x_3]$, la variable associée β (en rouge) est incrémentée et devient égale à 3. La condition permettant l'extension est vérifiée (ici, nous supposons f_e égal à 1) et l'extension est réalisée. Le solveur



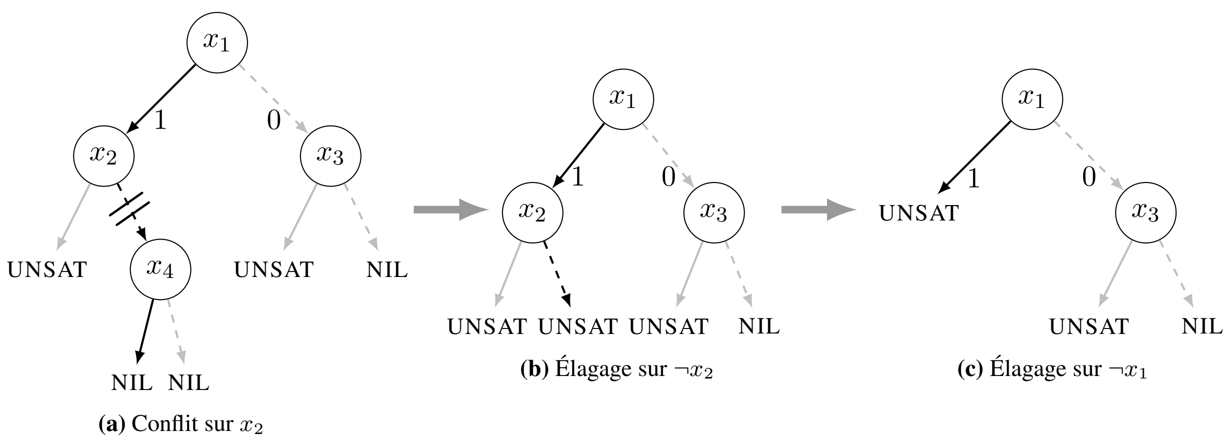
(a) Demande d'extension par le solveur S_3

(b) Extension acceptée par le MANAGER

FIGURE 6.5 – La figure de gauche représente l'arbre avant l'extension. Comme la valeur de β satisfait le critère d'extension, le MANAGER l'accepte et modifie l'arbre pour obtenir celle de droite.

3 envoi sa meilleure variable (x_5) et le cube $[\neg x_1, \neg x_3]$ est étendu avec cette variable en générant deux nouveaux cubes. Notons que la valeur β initiatrice de cette extension (en rouge) devient inutile car le nœud associé n'est plus une feuille. Le solveur S_3 est maintenant libre de demander au MANAGER un nouveau cube à résoudre. De plus, dans la prochaine étape, quelque soit le solveur demandant une extension, elle ne sera pas effectuée car le nombre de feuilles disponibles est maintenant égal à 4 (Nous avons supposé dans cette section que f_e est toujours égal à 1, nous discutons de ce facteur dans la section 6.3.1).

6.1.4 L'élagage



(a) Conflit sur x_2

(b) Élagage sur $\neg x_2$

(c) Élagage sur $\neg x_1$

FIGURE 6.6 – Élagage dans AMPHAROS

Comme nous utilisons la notion d'hypothèse, quand un cube s'avère être insatisfaisable, le solveur (celui prouvant l'insatisfaisabilité) calcule une clause conflit étant la négation d'un sous-ensemble des littéraux hypothèses. Cette information est transférée au MANAGER afin qu'il calcule un niveau de cou-

pure dans l’arbre. Ainsi, l’arbre est simplifié en conséquence. Remarquons qu’un solveur peut prouver directement l’insatisfaisabilité globale du problème lorsque la clause conflit calculée est vide. De plus, si les deux fils d’un nœud sont insatisfaisables alors ce nœud devient également insatisfaisable. Dans ce cas, ce nœud peut être supprimé et l’insatisfaisabilité est directement associée à la branche du parent. Bien sûr, ce processus est appliqué récursivement jusqu’à l’obtention d’un nœud possédant au moins un fils non insatisfaisable (figure 6.6).

6.2 L’échange des clauses

Dans cette section, nous examinons les deux façons d’échanger les informations dans AMPHAROS. Nous expliquons d’abord comment les clauses apprises par un solveur (*worker)* sont partagées avec les autres puis nous présentons une approche originale permettant de partager des littéraux unitaires locaux en prenant avantage de la structure de l’arbre géré par le MANAGER.

6.2.1 Le partage classique des clauses apprises

Il est bien connu que le partage des clauses apprises améliore sensiblement la performance des solveurs SAT parallèles (Hamadi *et al.* 2009c) (voir chapitre 4). Ici, les solveurs partagent également certaines clauses apprises. Cependant, les solveurs ne s’échangent pas directement les clauses entre eux mais ces dernières transitent via le MANAGER.

Au niveau des solveurs

À chaque fois qu’un solveur atteint un certain nombre de conflits (500 dans notre implémentation), il communique avec le MANAGER pour envoyer et/ou recevoir un ensemble de clauses. Les clauses devant être envoyées sont enregistrées dans une mémoire tampon qui est effacée après chaque communication avec le MANAGER. Les clauses possédant initialement un bon LBD (inférieur ou égal à 2) sont directement mises dans le tampon. D’autres clauses sont également ajoutées, comme dans (Audemard et Simon 2014), si elles participent à l’analyse d’un conflit. Cependant, comme nous ne pouvons pas partager autant de clauses que dans SYRUP, seules les clauses qui obtiennent un LBD dynamique inférieur ou égal à 2 avant d’être utilisées deux fois dans l’analyse d’un conflit sont partagées.

Afin de récupérer les clauses importées, les solveurs possèdent chacun trois tampons : `standby`, `purgatory` et `learnt`. Les clauses reçues sont d’abord stockées dans le `standby`. Dans ce tampon, les clauses ne sont pas attachées au solveur (Audemard *et al.* 2011). Tous les $4,000$ conflits, les clauses sont examinées : elles peuvent être transférées d’un tampon à un autre, être définitivement supprimées ou, enfin, être gardées dans le tampon courant.

Une clause du `standby` peut être transférée dans le `purgatory`. Contrairement au `standby` les clauses du `purgatory` sont attachées au solveur et participent à la propagation. Nous discutons du critère permettant de déplacer une clause du `standby` au `purgatory` dans la Sect. 6.3.1.

De la même manière, une clause du `purgatory` peut être transférée dans les `learnt` si elle est utilisée au moins une fois dans l’analyse d’un conflit. Le tampon temporaire `purgatory` est utilisé pour limiter l’impact des nouvelles clauses dans la stratégie de réduction des clauses apprises. Remarquons que le solveur SYRUP utilise une stratégie similaire (section 4.1.8, page 98).

La stratégie de nettoyage de ces deux tampons supplémentaires (`standby` et `purgatory`) dépend d’un compteur associé à chaque clause. Le compteur est incrémenté à chaque fois que les clauses sont

examinées. Si le compteur atteint un certain seuil (14 dans notre implémentation), la clause est supprimée. Notons que le compteur d'une clause est réinitialisé à chaque fois qu'elle change de tampon.

Au niveau du MANAGER

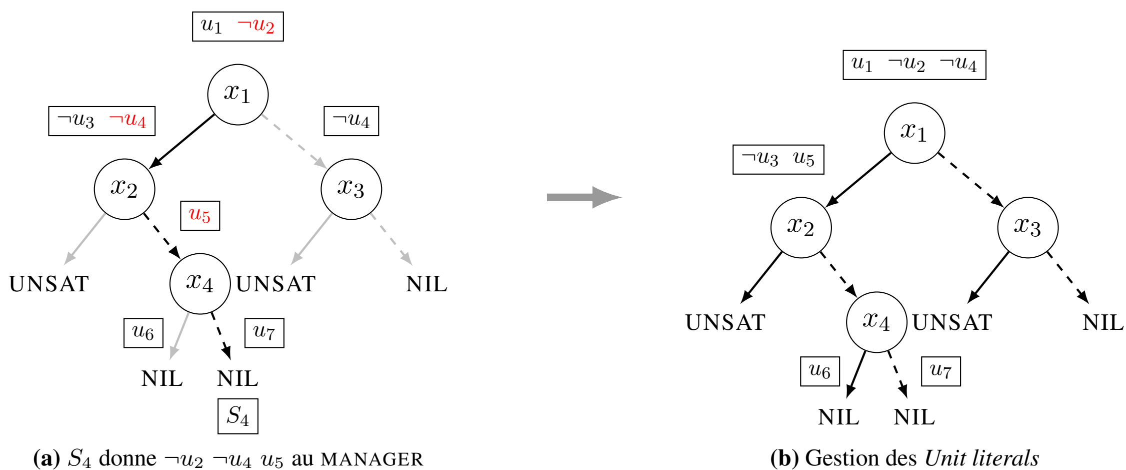
Le MANAGER gère les clauses apprises de tous les solveurs. Les clauses apprises sont stockées dans une queue et le MANAGER vérifie périodiquement si elles sont subsumées ou pas. En pratique, un seul cœur de calcul est dédié au MANAGER. De ce fait, vérifier toutes les clauses en même temps peut s'avérer très coûteux en temps de calcul et peut bloquer les communications entre le MANAGER et les solveurs. Afin d'éviter cette situation, le MANAGER calcule les clauses subsumées par paquet de $1,000$ et peut ainsi s'occuper entre temps des communications avec les solveurs. Le MANAGER garde uniquement les clauses apprises qui ne sont pas subsumées dans sa base et les envoient à chaque fois qu'un solveur les demande.

6.2.2 Les littéraux unitaires sous hypothèses

La seconde façon d'échanger des informations dans notre approche est de transférer les littéraux unitaires (qui sont propagés grâce au littéraux provenant des hypothèses) entre les solveurs et le MANAGER. Dans cette section, nous présentons d'où proviennent ces littéraux et comment ils sont échangés et gérés.

Au niveau des solveurs

Rappelons que chaque solveur travaille sous une hypothèse A (qui peut être vide) représentant le cube à résoudre (définition 4.8, section 4.2.5, page 108). Quand un littéral $\ell \notin A$ est propagé grâce à une sous-hypothèse $A' \subseteq A$, cette information peut être transmise au MANAGER afin d'être diffusée aux autres solveurs. Plus précisément, le solveur communique au MANAGER que ℓ peut être propagé avec A'. De l'autre côté, quand un solveur sélectionne une branche (ie un littéral ℓ') durant la transmission d'un cube, il va aussi recevoir un ensemble de littéraux unitaires associé à ℓ' afin de les propager. De ce fait, la transmission d'un cube (voir Sect. 6.1.2) contient ces messages additionnels.



(a) S_4 donne $\neg u_2$ $\neg u_4$ u_5 au MANAGER **(b)** Gestion des *Unit literals*

FIGURE 6.7 – Littéraux unitaires sous hypothèse dans AMPHAROS. La figure de gauche montre un arbre décoré avec les littéraux donnés par S_4 en rouge. Ces littéraux sont remontés, en utilisant deux règles (insatisfaisabilité pour u_5 et littéraux identiques pour $\neg u_4$) afin d'obtenir celle de droite.

Par conséquent, le MANAGER s'occupe de décorer l'arbre contenant les chemins de guidages par des ensembles de littéraux unitaires devant être propagés à chaque branche. La figure 6.7.a montre l'exemple d'un tel arbre. Quand un solveur S_4 demande une branche, il commence par récupérer l'ensemble des littéraux unitaires $\{u_1\}$. Il propage aussi $\neg u_2$ (en rouge) et donne cette information au MANAGER. Il choisit la branche x_1 et récupère le littéral $\neg u_3$ afin de le propager. De la même manière, il propage aussi $\neg u_4$ et apporte ce littéral unitaire sous hypothèses au MANAGER et ainsi de suite.

Comme nous allons le voir dans la partie expérimentation, les littéraux sous hypothèses sont très importants. Ce sont des clauses spéciales qui réduisent clairement l'espace de recherche d'une branche donnée. Par conséquent, le fait qu'un littéral ℓ peut être propagé à partir de A' est pris en compte dans le solveur concerné en ajoutant, dans une base dédiée, une clause composée de la négation de A' et du littéral ℓ' (cette base n'est pas la même que celle des clauses apprises (`learnt`) car elle n'est jamais nettoyée). Remarquons que quand $A' = \emptyset$, le littéral ℓ' est unitaire et est ajouté dans les littéraux unitaires du solveur.

Au niveau du MANAGER

Quand le MANAGER apprend qu'un littéral peut être propagé à partir d'un sous-ensemble de littéraux provenant d'une hypothèse, ce littéral est communiqué pendant la transmission du cube et ajouté dans la dernière branche du nœud associé à cette sous-hypothèse. Dans l'arbre, on peut remonter certains littéraux d'une branche dans d'autres branches plus hautes. Cette situation arrive soit quand une branche est prouvée insatisfaisable, soit quand les deux branches d'un même nœud possèdent le même littéral (Berre 2001a) (comme souligné dans la Fig. 6.7). Dans le premier cas (Cas 1), tous les littéraux de la branche non-insatisfaisable sont remontés dans la branche père (comme le littéral u_5). Dans le second cas (Cas 2), les littéraux apparaissant dans les deux branches d'un nœud sont remontés dans la branche père (c'est le cas du littéral $\neg u_4$). Ce processus est exécuté récursivement jusqu'à l'obtention d'un point fixe. Remarquons que quand la branche père n'existe pas (cela arrive quand les littéraux sont déplacés à partir d'une branche de la racine), alors ces littéraux sont prouvés unitaires.

Certaines de ces simplifications ont déjà été utilisées dans le passé comme techniques de prétraitement (*Failed Literal Probing*, section 2.3.7, page 53). L'originalité de notre approche est donc de faire de l'*inprocessing* (section 2.3.8, page 57) sur les sous-problèmes grâce aux informations échangées. Les techniques de prétraitement n'apportent toujours pas la même efficacité lorsqu'elles sont utilisées en *inprocessing*. Toutefois, des travaux récent ont obtenus de très bon résultats (section 2.3.8, *Learnt Clause Minimization*, Luo *et al.* (2017)). À notre connaissance, il existe peu de travaux sur les techniques d'*inprocessing* en parallèle. Cependant, nous pouvons citer les travaux de Wieringa et Heljanko (2013) (section 4.1.7, page 97) et ceux de Manthey *et al.* (2013). L'idée originale apportée par ces auteurs est d'utiliser un *thread* afin de simplifier les clauses apprises. Nous allons maintenant expliquer plus formellement les deux cas (Cas 1 et Cas 2).

Dans le premier cas (Cas 1), nous avons une des deux branches d'un nœud insatisfaisable et la branche non insatisfaisable contient le littéral unitaire sous hypothèse u_5. Formellement, soit Σ la formule initiale, nous avons :

$$\Sigma \wedge x_1 \wedge x_2 \vDash \bot$$
$$\Sigma \wedge x_1 \wedge \neg x_2 \vDash u_5$$

Ces deux relations de satisfaisabilité peuvent être simplifier en :

$$\Sigma \wedge x_1 \vDash u_5$$

Dans le deuxième cas (Cas 2), nous avons :

$$\Sigma \wedge x_1 \models \neg u_4$$
$$\Sigma \wedge \neg x_1 \models \neg u_4$$

Ces deux relations de satisfaisabilité peuvent être simplifier en :

$$\Sigma \models \neg u_4$$

6.3 Intensification vs Diversification

Quand plusieurs solveurs résolvent en concurrence un problème, ils peuvent faire des recherches redondantes. Identifier une telle situation permettrait de pouvoir modifier la stratégie des solveurs afin de diversifier leurs recherches. Toutefois, à cause du partage des clauses apprises entre les solveurs, explorer différents espaces de recherche pourrait être un handicap. Dans une telle situation, focaliser tous les solveurs sur le même espace de recherche pourrait être essentiel (intensification).

Ce paradigme, appelé dilemme d'intensification/diversification, a déjà été étudié dans le contexte des solveurs SAT de type *portfolio*. Il peut être traité soit statiquement, en utilisant plusieurs solveurs avec des stratégies opposées (Audemard *et al.* 2012, Hamadi *et al.* 2009c, Roussel 2011), soit dynamiquement, en modifiant la stratégie des solveurs pendant la recherche. Cependant, savoir quand la recherche d'un solveur doit être intensifiée ou diversifiée n'est pas facile et seulement quelques publications essaient de traiter ce problème (Guo *et al.* 2010, Guo et Lagniez 2011a). Dans (Guo *et al.* 2010), une architecture maître/esclave est proposée. Les maîtres tentent de résoudre le problème original (assurant la diversification), tandis que les esclaves intensifient la stratégie de leur maître. Dans (Guo et Lagniez 2011a), une mesure pour estimer le degré de la redondance entre deux solveurs est présentée. Elle considère que deux solveurs sont proches quand ils ont approximativement le même vecteur de polarité (*phase-saving*). Le processus de diversification consiste alors à modifier la manière de choisir la polarité des variables de décision.

À notre connaissance, même si beaucoup de solveurs diversifient la recherche (chapitre 4), aucun critère permet d'identifier si plusieurs solveurs exécutent une recherche redondante mis à part la mesure sur la polarité dans Guo et Lagniez (2011a). Dans nos travaux, nous étudions un nouveau critère essayant d'identifier les recherches redondantes grâce aux partages d'informations.

6.3.1 Évaluation du degré de redondance

Nous proposons de mesurer le degré de redondance en prenant en compte le nombre de clauses redondantes partagées entre les solveurs. Nous utilisons une liste afin de mémoriser depuis le début le nombre de clauses reçues (st_r) et une autre afin de mémoriser le nombre de clauses gardées (st_k). Les clauses gardées sont celles qui n'ont pas été supprimées durant la vérification des clauses subsumées. Quand un solveur revient vers le MANAGER pour partager des clauses (tous les $1,000$), le nombre de clauses reçues (resp. gardées) depuis le début est sauvegardé dans st_r (resp. st_k) par le MANAGER.

La *redundancy shared clauses measure*, en bref $rscm$, est définie pour une étape t suivant un intervalle glissant de taille m ($20,000$ dans nos expérimentations) comme le ratio entre le nombre de clauses reçues durant les $t - m$ mises à jour de st_r et le nombre de clauses gardées durant ce même temps. Plus précisément, nous avons $\forall j < 0$, $st_r[j] = st_k[j] = 0$:

$$\begin{aligned} rscm_t &= \frac{st_r[t] - st_r[t-m]}{st_k[t] - st_k[t-m]}, \text{ si } st_k[t] - st_k[t-m] \neq 0 \\ rscm_t &= st_r[t] - st_r[t-m], \text{ sinon} \end{aligned} \tag{6.1}$$

Tout d'abord, notons que lorsque plusieurs solveurs travaillent sur le même espace de recherche, il y a une forte probabilité que les clauses apprises par les différents solveurs soient redondantes. Cela signifie que le nombre de clauses subsumées est important, et donc, d'avoir un $rscm$ élevé. Inversement, lorsque les solveurs sont diversifiés dans l'espace de recherche, il y a une forte probabilité d'avoir des clauses non redondantes, et donc, d'avoir un $rscm$ faible. Par conséquent, un $rscm$ faible indique que le solveur doit intensifier la recherche, tandis qu'une valeur $rscm$ élevée signifie que les solveurs doivent diversifier leur recherche.

Il y a plusieurs façons de diversifier ou d'intensifier les solveurs (clauses apprises, heuristiques des solveurs, ...). Dans AMPHAROS, nous choisissons de résoudre le dilemme d'intensification/diversification en contrôlant deux critères : la manière dont l'arbre est étendu (voir Sect. 6.1.3) et le nombre de clauses passant du `standby` aux `purgatory` (voir Sect. 4.1). Ainsi, pour nous, diversifier (resp. intensifier) la recherche consiste à augmenter (resp. diminuer) ces deux paramètres.

Peu de clauses subsumées ($rscm$ est petit)	**Beaucoup de clauses subsumées** ($rscm$ est grand)
Réduire l'extension Augmenter les clauses importées	Favoriser l'extension Limiter les clauses importées
Intensification	**Diversification**

L'extension guidée par le $rscm$

Tout d'abord, remarquons que chaque chemin de la racine à une feuille représente un ensemble unique de littéraux qui divise l'espace de recherche d'une manière déterministe. Ainsi, plus l'arbre est grand, plus la probabilité d'exécuter deux solveurs dans deux sous-problèmes distincts augmente. Afin de contrôler cette croissance, nous définissons le facteur d'extension :

$$fe_t = \frac{1000}{{rscm_t}^3} \tag{6.2}$$

Rappelons que ce facteur d'extension est utilisé afin d'accroître ou de diminuer l'extension de l'arbre et est associé au nombre de cubes disponibles. Par conséquent, plus la valeur $rscm_t$ est petite (resp. grande), plus la valeur fe sera grande (resp. petite), et donc plus l'extension de l'arbre sera lente (resp. rapide). Notons que le ${rscm_t}^3$ (au cube) permet de diminuer fe rapidement, tandis que la division (par 1000) permet de le limiter au cas où les solveurs seraient en concurrence. Afin d'empêcher l'arbre de s'étendre trop, nous avons également limité la valeur de fe a un maximum de 10.

Du `standby` au `purgatory`

Quand une clause est reçue par un solveur, elle peut être subsumée par celles déjà présentes. Ainsi, il semble naturel que le nombre de clauses acceptées par un solveur (celles passant du `standby` au `purgatory`) augmente (resp. diminue) quand la valeur $rscm$ diminue (augmente). Autrement dit, plus (resp. moins) nous avons de clauses subsumées, moins (resp. plus) les solveurs ajoutent des clauses reçues.

Dans AMPHAROS, les clauses fraîchement reçues ne sont pas directement attachées au solveur. Ainsi, nous devons choisir un critère de sélection indépendant afin d'activer ces clauses. Pour contrôler la

quantité de clauses passant du `standby` au `purgatory`, nous utilisons la notion de psm ((Audemard *et al.* 2011)) et déjà utilisée dans le solveur de type *portfolio* PENELOPE (section 4.1.6, page 96, Audemard *et al.* (2012)). Rappelons que le psm (section 2.3.5, page 49) d'une clause représente le nombre de littéraux qui sont affectés à vrai par le vecteur de polarité (*phase-saving*).

Ensuite, afin d'augmenter/diminuer le nombre de clauses attachées (et alors transférées) dans le `purgatory`, un critère fusionnant les valeurs psm et $rscm$ est proposé. Ce critère est justifié par l'observation de (Audemard *et al.* 2011). Les auteurs montrent expérimentalement que les clauses possédant un petit psm ont plus de chance d'entrer en conflit ou d'être utilisées pendant la recherche. De ce fait, dans notre méthode, une clause est autorisée à passer du `standby` au `purgatory` quand sa valeur psm est inférieure ou égale à $\lfloor \frac{psm_{max}}{rscm_t} \rfloor$, où psm_{max} correspond au psm maximum accepté (fixé à 6 dans nos expérimentations). En conséquence, les clauses avec un psm de zéro sont systématiquement acceptées quelles que soient la valeur $rscm$. Tandis que les clauses possédant une valeur psm élevée seront acceptées si et seulement si il y a peu de clauses subsumées au niveau du MANAGER.

6.4 Évaluation expérimentale

Nous évaluons AMPHAROS sur les 100 instances du *parallel track* de la SAT-RACE 2015 (SAT). Durant cette compétition, 53 (resp. 33) instances ont été prouvées satisfaisables (resp. insatisfaisables) par au moins un des solveurs et 14 instances sont restées non résolues. Les expérimentations conduites dans cet article sont réalisées sur des bi-processeurs Intel XEON X5550 4 cœurs à 2.66 GHz avec 8Mo de cache et 32 Go de RAM, sous Linux CentOS 6 (pour un total de 64 cœurs). La limite de temps alloué pour résoudre une instance est de 1200 secondes (temps réel). Pour les expérimentations faites sur 64 cœurs, nous utilisons donc deux machines. Tous les fichiers de *log*, des diagrammes de dispersion et des cactus additionnels sont disponibles à l'adresse `http://www.cril.univ-artois.fr/ampharos`.

6.4.1 Gestion des communications

Comme dans AMPHAROS un grand nombre de messages sont échangés entre le MANAGER et les solveurs, la gestion des communications doit être efficace. Ainsi, nous avons choisi la librairie *open source* Open MPI (pour *Message Passing Interface implementation*) afin de gérer les communications.

Le goulot d'étranglement imposé par le fait que le MANAGER doit à la fois communiquer avec tous les solveurs et calculer les clauses subsumées a été un problème majeur. Afin d'éviter que les solveurs attendent trop longtemps sans travailler, un tourniquet associé à une écoute non-bloquante des solveurs par le MANAGER a été mise en place. De plus, comme nous identifions les clauses subsumées reçues et que ceci peut être coûteux, nous ne les traitons pas dés leur réception mais petit à petit par paquet de $1,000$ clauses.

6.4.2 Configuration

AMPHAROS est un outil modulaire qui permet d'ajouter facilement de nouveaux types de solveurs. Pour ces expérimentations, trois solveurs SAT séquentiels ont été utilisés : GLUCOSE (Audemard et Simon 2009b), MINISAT (Sörensson et Eén 2009) et MINISATPSM (Audemard *et al.* 2011). Seuls quelques modifications mineures de ces solveurs ont été nécessaires. Afin de gérer les interactions avec le MANAGER, tous les solveurs doivent implémenter une interface en C++. Cette interface permet de regrouper les méthodes engendrant des communications ou celles faisant intervenir des ajouts dans les solveurs (les mémoires tampons `standby` et `purgatory`). Le noyau des solveurs a également été modifié afin

d'éviter de réinitialiser certaines heuristiques à chaque fois qu'un cube doit être résolu (*restart*, nettoyage de la base des clauses apprises, ...). Par ailleurs, comme pour la version de GLUCOSE présentée dans (Audemard *et al.* 2013a), quand un solveur redémarre, il n'a pas besoin de faire un saut au niveau de décision 0, mais au niveau de la dernière hypothèse. Les clauses transférées du `purgatory` au `learnt` sont simplement incorporées dans la base des clauses apprises du solveur comme si elles étaient apprises par lui-même.

6.4.3 Résultats

L'évaluation expérimentale est divisée en trois parties. Premièrement, nous évaluons les différentes approches d'AMPHAROS. Puis, nous étudions l'extensibilité de notre solveur. Finalement, nous comparons AMPHAROS avec quelques solveurs de l'état de l'art.

L'impact de chaque composant

Les bénéfices des trois composants optionnels (décomposition via un arbre, (**Tree**), partage des clauses (**C**), et échange des littéraux unitaires (**UL**)) ont été étudiés expérimentalement. À cette fin, plusieurs versions d'AMPHAROS ont été exécutées sur 64 cœurs. Ces expériences, présentées dans la figure 6.8 et le tableau 6.1, montre une amélioration progressive lorsque chacune de ces options a été prise en compte simultanément. Plusieurs observations peuvent être déduites de ce *cactus plot* (figure 6.8).

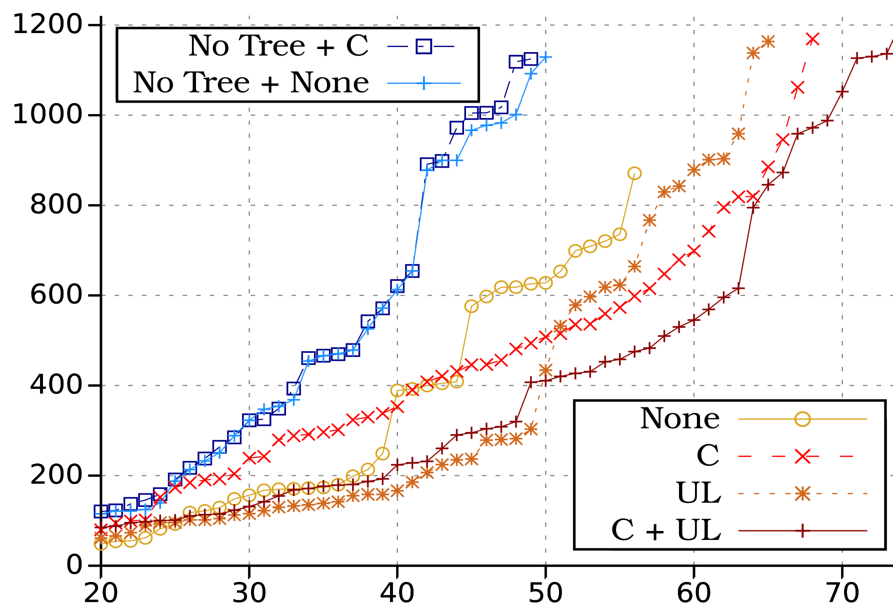


FIGURE 6.8 – Évaluation des composants du solveur AMPHAROS (*cactus plot*).

Premièrement, quelle que soit la combinaison des options utilisées, AMPHAROS est plus efficace avec la décomposition via un arbre. Les versions concurrentielles (pouvant être considérées comme *portfolio*) (**No Tree**) avec (**C**) ou sans (**none**) le partage des clauses, résolvent systématiquement moins d'instances que les versions de type « diviser pour mieux régner ». Cela montre l'importance de la manière de résoudre le dilemme intensification/diversification en utilisant une décomposition via un arbre dans AMPHAROS (**Tree Yes**).

Division (Tree)	Partages	SAT	UNSAT	Total
Oui	C + UL	**49**	**25**	**74**
Oui	C	47	21	68
Oui	UL	47	18	65
Oui	None	41	15	56
Non	C	43	6	49
Non	None	44	6	50

TABLE 6.1 – Évaluation des composants du solveur AMPHAROS. Nombre d'instances résolues pour chaque composant.

Deuxièmement, Les résultats montrent l'importance du partage des informations entre les solveurs quand les cubes sont activés. La version d'AMPHAROS qui n'échange aucune information résout systématiquement moins d'instances que celles qui en échangent ((**C**) et/ou (**UL**)). Quand nous comparons séparément ces deux types d'échange, nous observons que le partage des clauses apprises permet d'améliorer les résultats sur les problèmes insatisfaisables (comme attendu). Toutefois, activer cette option rend le solveur plus lent sur les problèmes faciles (résolus en moins de 600 secondes). Ceci peut être expliqué par le fait que les communications engendrées par les clauses partagées peuvent diminuer la vitesse de notre solveur sur ces instances « faciles ».

Finalement, nous mettons en évidence une synergie très positive entre les types d'échange. Même si le partage des clauses peut dramatiquement réduire les performances du solveur sur les instances faciles, la combinaison de ce composant avec l'échange des littéraux unitaires sous hypothèses (**C + UL**) permet de donner l'amélioration la plus significative à la fois en nombre d'instances résolues et sur le cactus 6.8. À partir de maintenant, quand nous parlerons d'AMPHAROS, il s'agira de la version **C + UL** avec division de l'arbre.

Évaluation de l'extensibilité

Afin d'évaluer l'extensibilité d'AMPHAROS, nous le lançons sur 8, 16, 32 et 64 cœurs. La Fig. 6.10 nous donne le nombre d'instances résolues par rapport au temps sur ces différentes configurations. Ce cactus montre clairement que notre approche est très extensible. La version sur 64 cœurs résout 49 SAT et 25 UNSAT, ce qui est 15%, 45% et 70% fois plus d'instances que celles sur, respectivement, 32 (44 SAT et 20 UNSAT), 16 (36 SAT et 15 UNSAT) et 8 (33 SAT et 11 UNSAT) cœurs. De plus, nous montrons aussi l'efficacité de notre solveur dans un environnement plus distribué avec 8 ordinateurs de 8 cœurs pour un total de 64 cœurs (courbe 8×8 de la figure 6.10). Cette version résout 46 instances satisfaisables et 24 insatisfaisable. Remarquons que les résultats changent car la configuration des machines est différente. Cependant, cela montre que les performances restent acceptables dans un environnement distribuée.

AMPHAROS vs l'état de l'art

Nous avons choisi d'évaluer notre approche avec les trois meilleurs solveurs du *parallel track* de la SAT-RACE 2015 (SAT) sur 32 cœurs puis avec deux solveurs compatibles avec une configuration distribuée sur 64 cœurs afin de se comparer avec la majorité des travaux existants. Les trois solveurs de la

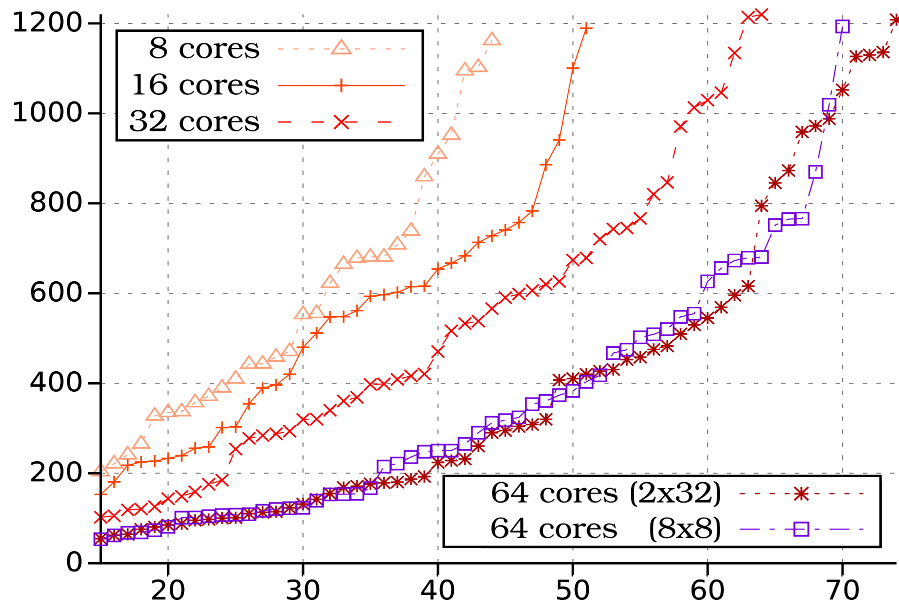


FIGURE 6.9 – Évaluation de l'extensibilité du solveur AMPHAROS sur 8, 16, 32 et 64 cœurs de calcul

compétition (suivant leur rang) sont : SYRUP (Audemard et Simon 2014), TREENGELING et PLINGELING (Biere 2013). Nous ne pouvons les lancer que sur 32 cœurs car ces solveurs ne sont pas distribués et nous n'avons pas de processeur contenant 64 cœurs mais uniquement deux machines de 32 cœurs. De ce fait, nous comparons aussi notre solveur avec d'autres solveurs SAT distribués sur 64 cœurs : le solveur parallèle de type *work stealing* DOLIUS (Audemard *et al.* 2014b) et le solveur de type *portfolio* HORDESAT (Balyo *et al.* 2015).

Solveur	#thr.	SAT	UNSAT	Total
AMPHAROS	32	**44**	20	**64**
SYRUP	32	36	**26**	62
TREENGELING	32	38	20	58
PLINGELING	32	31	**26**	57
AMPHAROS	64	**49**	**25**	**74**
HORDESAT	64	33	24	57
DOLIUS	64	33	17	50

TABLE 6.2 – Résultats de chaque solveur suivant le nombre de *threads* (**#thr.**).

Considérons tout d'abord, les expérimentations sur 32 cœurs. AMPHAROS résout plus d'instances que les autres solveurs (tableau 6.2). Il est le meilleur solveur sur les instances satisfaisables et résout autant d'instances insatisfaisables que TREENGELING (qui lui aussi est un solveur de type « diviser pour mieux régner »). Par rapport à SYRUP et PLINGELING, notre solveur est meilleur sur les instances satisfaisables mais moins efficace sur celles insatisfaisables. Ceci peut partiellement s'expliquer par le fait qu'AMPHAROS résout essentiellement les problèmes insatisfaisables en réfutant complètement l'arbre (*i.e.* en fermant toutes les branches). En conséquence, il semble que sur 32 cœurs, nous n'avons pas assez de *workers* pour atteindre ce but dans le temps imparti.

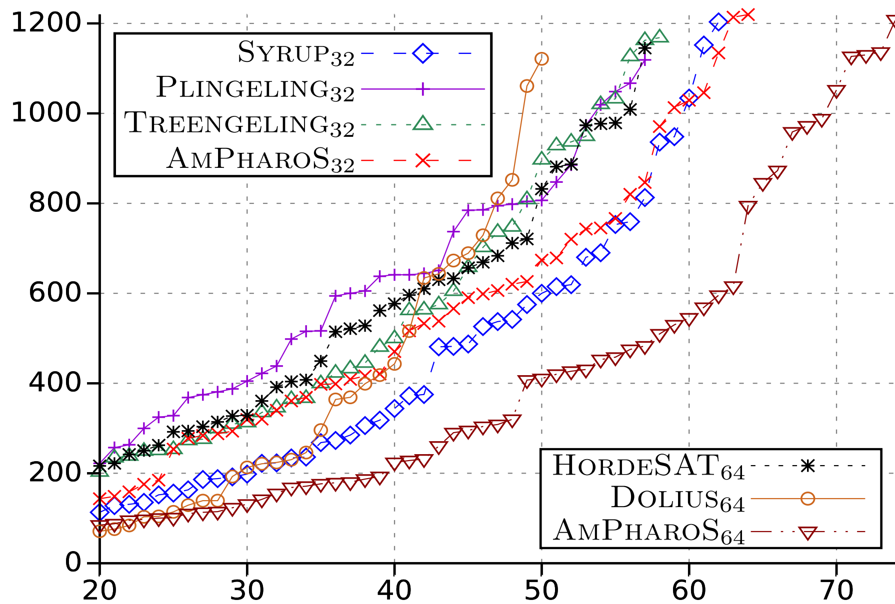


FIGURE 6.10 – Comparaisons d'AMPHAROS vs l'état de l'art des solveurs SAT parallèles (*cactus plot*).

En considérant les temps réels du cactus 6.10, nous pouvons observer qu'AMPHAROS est plus rapide que TREENGELING et PLINGELING mais plus lent que SYRUP. Nous pouvons expliquer ceci par le fait que SYRUP résout plusieurs instances d'une même famille très rapidement (la famille des instances `6s`).

Dans un second temps, nous pouvons observer sur 64 cœurs que notre approche est très compétitive. AMPHAROS est significativement meilleur que DOLIUS et HORDESAT. De plus, il est important de noter que, durant la compétition, SYRUP (le gagnant du *parallel track*) a utilisé seulement 32 cœurs sur les 64 disponibles. En conséquence, il est possible d'admettre, par transitivité, qu'AMPHAROS serait plus efficace que TREENGELING et PLINGELING sur 64 cœurs. Plus important, nous pouvons voir sur le cactus 6.10 qu'AMPHAROS est vraiment efficace car il résout plus d'instances plus rapidement.

Notons que ces solveurs sont indéterministes. Pour être juste, nous avons fait tourner une seule fois chaque solveur et reporté les résultats obtenus (comme cela est fait à la SAT RACE 2015). Pour conclure cette section, nous allons maintenant effectuer une évaluation expérimentale de la valeur $rscm$.

Impact de la valeur $rscm$

Dans cette section, nous évaluons l'impact de la valeur $rscm$ sur les performances globales d'AMPHAROS. À cette fin, nous avons sélectionné une ensemble représentatif d'instances provenant de la compétition SAT de 2015. Nous testons quatre versions d'AMPHAROS avec différente valeur $rscm$ (1,3,5 et 10) et nous les comparons avec notre version dynamique du $rscm$. Rappelons que le $rscm$ à un impact à la fois sur l'extension de l'arbre et sur la quantité de clauses échangées. De plus, comme nous l'avons mentionné dans la section 6.3.1, le facteur d'extension fe est fixé à 10 quand $rscm > \sqrt[3]{100}$. Ainsi, l'impact entre $rscm = 5$ et $rscm = 10$ est seulement dû à la quantité de clauses partagées.

Le tableau 6.3 présente les résultats obtenues sur un ensemble représentatif d'instances. Chaque ligne correspond à une instance, avec sa satisfaisabilité (première et deuxième colonnes). Les quatre colonnes suivante donne le temps réel (en secondes) nécessaire pour résoudre l'instance en fonction de la valeur $rscm$ choisie . Celle-ci est soit statique (1, 3, 5, et 10) ou bien dynamique (D). Les statistiques les plus à droite sont obtenues par la version dynamique du $rscm$.

Le tableau 6.3 montre que la résolution des instances n'a pas le même comportement en fonction de la valeur $rscm$ choisie. Quelques problèmes ont besoin de faire plus d'extension comme pour l'instance `jgiraldezlevy` et d'autres ont besoin d'étendre moins l'arbre et échanger plus de clauses comme l'instance `minandmaxor128`. Il est aussi important de noter que quelques instances sont imprévisibles comme celle nommée `004-80-8`. Quand nous regardons la colonne du tableau représentant l'ajustement dynamique, nous observons que celui-ci est en moyenne souvent très proche de la meilleure valeur $rscm$. Les différences de performances proviennent du fait que notre solveur à besoin de temps afin de tendre vers la bonne valeur $rscm$

Informations		**Temps w.r.t. rscm**					**Statistiques rscm**			
Nom	**Solution**	**1**	**3**	**5**	**10**	**D**	**min**	**max**	**avg**	**med**
hitag2-10-60-0-65	UNSAT	563	173	120	127	304	1.20	2.36	2.11	2.28
jgiraldezlevy.109	UNSAT	544	243	214	154	260	1.60	4.32	3.76	4.12
minandmaxor128	UNSAT	788	IN	IN	IN	972	1.09	1.37	1.25	1.23
jgiraldezlevy.33	SAT	776	339	386	169	288	1.63	4.58	3.32	3.82
56bits-12.dimacs	SAT	114	168	180	395	101	1.25	1.60	1.43	1.46
004-80-8	SAT	248	412	16	264	110	1.41	4.64	3.10	3.09

TABLE 6.3 – Étude de la valeur $rscm$ d'AMPHAROS.

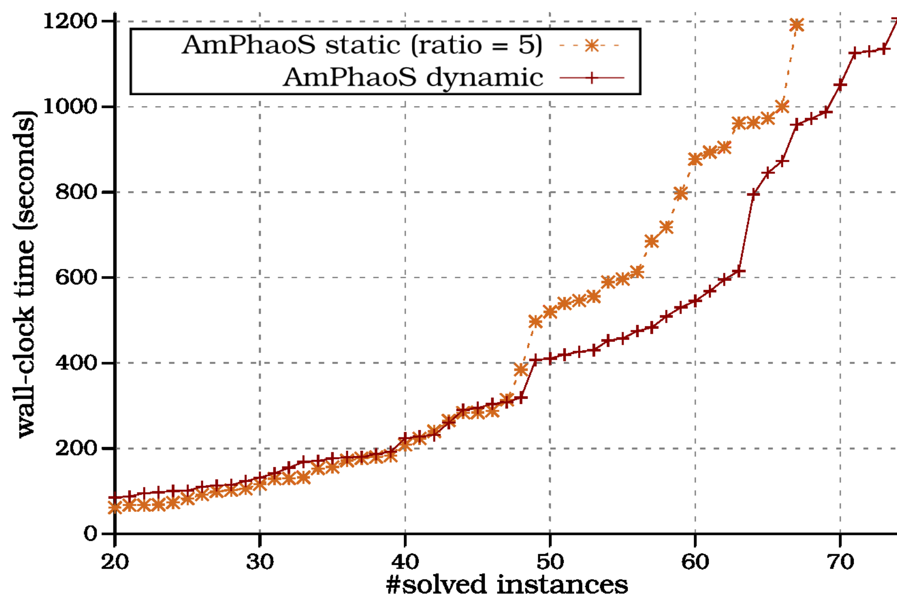


FIGURE 6.11 – Étude de la valeur $rscm$ d'AMPHAROS (*cactus plot*). Valeur $rscm$ statique égale à 5 contre un $rscm$ dynamique.

En complément, nous avons lancé AMPHAROS avec une valeur $rscm$ statique égale à 5 contre celle dynamique par défaut dans AMPHAROS (figures 6.11 et 6.12). La limite de temps alloué pour résoudre une instance est de 1200 secondes (temps réel). Pour ces expérimentations faites sur 64 cœurs, nous utilisons deux machines. Elles sont réalisées sur les 100 instances du *parallel track* de la SAT-RACE 2015 (SAT). Nous montrons que notre solveur est alors plus performant avec celle dynamique.

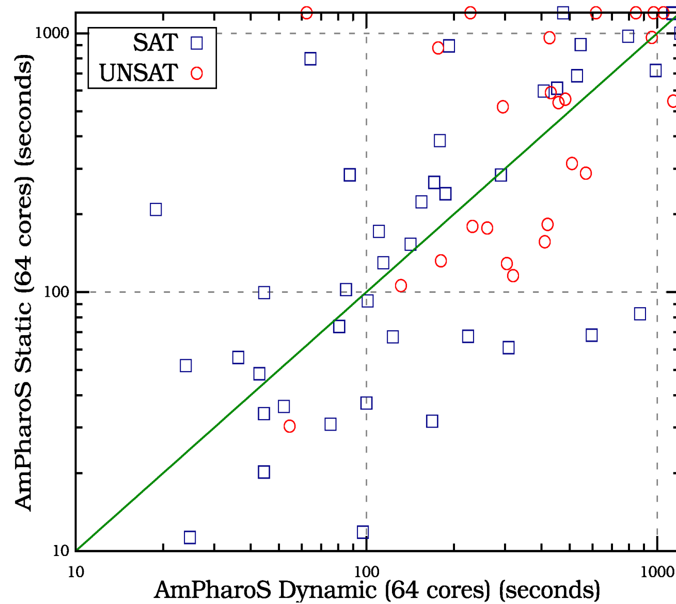


FIGURE 6.12 – Étude de la valeur $rscm$ d'AMPHAROS (*scatter plot*). Valeur $rscm$ statique égale à 5 contre un $rscm$ dynamique.

6.5 Conclusion

Nous avons proposé un nouveau solveur SAT parallèle, destiné à être exécuté sur de nombreux cœurs et basé sur le paradigme « diviser pour mieux régner ». Notre solveur permet de partager deux sortes de clauses, les classiques clauses apprises et d'autres liées à la division du problème initial. Nous proposons de mesurer à quelle point les solveurs font du travail redondant en comptant le nombre de clauses partagées et subsumées. Cette mesure nous permet d'ajuster dynamiquement la recherche afin de l'intensifier ou de la diversifier et ainsi résoudre en partie ce dilemme. Les expérimentations montrent des résultats très prometteurs.

Cependant, nous avons observé une anomalie dans AMPHAROS. Plus précisément, dans la version concurrentielle d'AMPHAROS, c'est-à-dire, sans division en sous-problème. Dans la figure 6.8 de la section 6.4.3, l'impact de chaque composant d'AMPHAROS est étudié. Nous pouvons alors observer que le mode concurrentiel avec partage des clauses (**No Tree + C** dans la figure) n'obtient pas les résultats attendus. Normalement, ces résultats devraient être bien meilleurs que ceux sans partage de clause (**No Tree + None**). En effet, l'échange des clauses dans les solveurs SAT parallèles à prouvé son efficacité et cela est bien connu dans la littérature (chapitre 4).

Nous pensons que cela est dû à une congestion du réseau. En effet, rappelons qu'AMPHAROS partage les clauses d'une manière centralisée (définition 3.32). Or, ce partage implique un plus grand nombre de données à travers le réseau qu'un partage d'informations décentralisée (définition 3.33). Cela provient du fait que les clauses sont d'abord envoyées à un maître puis redistribuées aux solveurs. Dans un partage d'informations décentralisée, les clauses sont directement partagées de solveur à solveur sans passé par un maître. Afin de remédier à ce problème, nous avons donc décider d'étudier différentes manières d'échanger les informations dans un solveur *portfolio* via les modèles de programmations parallèles (section 3.6.2).

L'objectif de travaux futurs est ainsi de corriger la congestion du réseau d'AMPHAROS. Comme il est très complexe de programmer plusieurs modèles de programmations parallèles à partir d'un solveur déjà

complexe comme AMPHAROS (plus de 10000 lignes de code), nous avons décidé de refaire un nouveau solveur *portfolio* à partir du solveur SYRUP afin d'étudier ces modèles de programmation. Le prochain chapitre (le 7) est donc consacré à ces contributions et expose un solveur *portfolio* nommé D-SYRUP qui est destiné à l'étude des modèles de programmation. Nous allons alors montrer que ces modèles jouent un rôle très important dans l'efficacité des solveurs distribués. Pour finir, nous allons présenté des travaux basées sur AMPHAROS.

6.5.1 MAPLEAMPHAROS : Le ratio de propagations par décision

▷ Nejati, Newsham, Scott, Liang, Gebotys, Poupart, et Ganesh (2017)
▷ Mémoire distribuée

Les auteurs Nejati *et al.* (2017) ont modifié l'heuristique permettant de choisir les variables de l'arbre de division d'AMPHAROS. Pour rappel, nous utilisons à cette fin l'heuristique EVSIDS dans AMPHAROS. L'idée originale apportée par ces auteurs est d'utiliser une heuristique basée sur un ratio de propagations à la place de EVSIDS. Pour cela, ils se sont inspirés des solveurs *look-ahead*. Plus précisément, quand un *worker* a atteint sa limite de conflit pour potentiellement changer le sous-problème sur lequel il travail, le *worker* choisit d'envoyer au MANAGER la variable qui à causée le plus grand nombre de propagations par décision (le ratio de propagation). Plus précisément, quand une variable v est décidée, MAPLEAMPHAROS compte le nombre de propagations unitaires notée $nbPropagations(v)$ effectué lors de cette décision. Ainsi, durant la durée de travail d'un *worker* sur un sous-problème (pendant un certain nombre de conflits), à chaque fois que la variable v est décidée, $nbPropagations(v)$ est mis à jour en lui ajoutant les propagations unitaires associées et le nombre de décisions de la variable v noté $nbDecisions(v)$ est incrémenté. Par la suite, le ratio de propagation d'une variable $PRSH(v)$ est calculé suivant :

$$PRSH(v) = \frac{nbPropagations(v)}{nbDecisions(v)}$$

$PRSH(v)$ est donc calculé quand un *worker* a fini de travailler sur un sous-problème ou que ce dernier est prouvé insatisfaisable. Si ce sous-problème est insatisfaisable, le *worker* prend un autre sous-problème. En revanche, dans le cas contraire, il peut soit recommencer à résoudre le même sous-problème, soit en prendre un nouveau déjà existant, soit l'étendre afin d'en créer deux nouveaux via une variable. Remarquons que, peu importe ces situations, le ratio n'est jamais remit à zéro. Quand une extension est effectuée, la variable ayant le plus grand ratio est choisie. Grâce à cette heuristique, les auteurs ont amélioré significativement les performances de leur solveur.

Chapitre

7

D-SYRUP : Un solveur « *portfolio* » distribué

Sommaire

7.1 Étude expérimentale de SYRUP 156
7.1.1 Instances résolues 156
7.1.2 Extensibilité 157
7.2 L'architecture distribuée 160
7.2.1 AMPHAROS 160
7.2.2 Interblocages 161
7.2.3 Les cycles de communications 161
7.2.4 Phase de recherche et phase de communications 162
7.2.5 Objectifs 162
7.3 Le modèle de programmation pur par passage de messages 162
7.3.1 Désavantages 163
7.3.2 Avantages 164
7.3.3 Utilisations 164
7.3.4 Expérimentations 164
7.4 Le modèle de programmation partiellement hybride 166
7.4.1 Avantages et désavantages 166
7.4.2 Utilisation 167
7.4.3 Implémentation 168
7.4.4 Expérimentations 168
7.5 Le modèle de programmation complètement hybride 169
7.5.1 Méthode 170
7.5.2 Le problème des interblocages sur le réseau 171
7.5.3 Expérimentations 171
7.6 Expérimentations 172
7.6.1 Comparaison des différents modèles de programmation 172
7.6.2 Évaluation de D-SYRUP 174
7.7 Conclusion 177

La programmation d'un solveur SAT parallèle partageant des données sur le réseau n'est pas si simple. Notamment, ces données sont principalement des clauses apprises par les solveurs et doivent donc être échangées à n'importe quel moment durant la recherche (chapitre 4). Qui plus est, il existe différentes manières de programmer les communications, quelles soient réalisées entre plusieurs machines ou entre plusieurs *threads* sur une même machine (chapitre 3). Plusieurs techniques de programmations parallèles sont alors regroupées en ce qui est communément appelées les modèles de programmations parallèles (section 3.6.2). Ce chapitre a pour objectif de présenter une étude des modèles de programmation parallèle dans le cadre de SAT et d'apporter une solution novatrice aux possibles congestions du réseau dans un environnement massivement distribué. Pour réaliser ces contributions, nous utilisons comme base le solveur *multi-thread* SYRUP (section 4.1.8, page 98). Ces travaux ont donnés lieu à une publication internationale (Audemard *et al.* 2017).

Dans ce chapitre, nous réalisons d'abord une étude expérimentale du solveur *multi-thread* SYRUP (section 7.1) afin d'exposer son extensibilité (définition 3.21, section 3.5, page 74). Ensuite, nous exposons ce qu'implique l'utilisation d'architectures distribuées dans le cadre de SAT et présentons nos objectifs (section 7.2). Puis, nous évaluons deux modèles de programmations distribuées existants et déjà utilisés dans des solveurs SAT. Le premier est nommé le modèle de programmation pur par passage de message (section 7.3) et est utilisé par AMPHAROS (chapitre 6) et TOPOSAT (section 4.1.10, page 100). Le deuxième est appelé le modèle partiellement hybride (section 7.4) et est utilisé par HORDESAT (section 4.1.11, page 4.1.11). Nous montrons que ces schémas ont un impact direct sur les performances des solveurs. De ce fait, nous introduisons un modèle complètement hybride à partir du solveur SYRUP qui n'a, à notre connaissance, jamais été appliqué à SAT (section 7.5). Avant de conclure (section 7.7) et d'apporter quelques perspectives, nous comparons expérimentalement les différents modèles de programmation ainsi qu'une version distribuée de SYRUP nommée D-SYRUP (basée sur le modèle de programmation complètement hybride) contre deux solveurs de l'état de l'art (section 7.6).

7.1 Étude expérimentale de SYRUP

Un point clé dans l'efficacité de SYRUP provient de la manière de sélectionner les clauses à partager ainsi que l'utilisation d'une structure de donnée dédiée à la gestion de ce partage. Ces deux caractéristiques se révèlent nécessaires afin d'éviter une surcharge des unités de calcul. Même si SYRUP est extensible jusqu'à 32 cœurs de calcul, il n'est pas initialement conçu pour fonctionner avec un nombre très élevé de cœurs. Durant la compétition SAT de 2015, il a seulement utilisé 24 des 48 cœurs disponibles. En particulier la quantité de clauses échangées entre les unités de recherche, ainsi que le nombre de clauses que les solveurs doivent gérer, augmentent en fonction du nombre de cœurs. Afin de réduire l'impact d'une telle quantité d'information à partager, le nombre de clauses pouvant être échangées est limité. Cette limite dépend du nombre de cœurs utilisés et est fixée empiriquement. Dans ces travaux, nos résultats montrent que partager trop de clauses ralentit dramatiquement les solveurs.

Afin d'évaluer les performances et l'extensibilité (section 3.5) du solveur *multi-thread* SYRUP (section 4.1.8), nous l'exécutons sur la totalité des 100 instances du *parallel track* de la SAT Race 2015 en utilisant 1, 2, 4, 8, 16 et 32 cœurs. Dans ces expérimentations, nous fixons le temps réel à 1800 secondes pour toutes les instances. Un ordinateur de 32 cœurs avec 256GB de mémoire RAM a été utilisé, c'est un *quad-processor Intel XEON X7550*.

7.1.1 Instances résolues

Comme le montre la figure 7.1 et la table 7.1, plus nous augmentons le nombre de cœurs, plus nous avons d'instances résolues et celles-ci sont résolues de plus en plus rapidement. La version séquentielle de SYRUP, GLUCOSE, résout 26 instances (15 satisfaisables et 11 insatisfaisables). Avec 8 cœurs, SYRUP résout 56 instances (31 satisfaisables et 25 insatisfaisables) et avec 32 cœurs, il résout 70 instances (42 satisfaisables et 28 insatisfaisables). Le nombre d'instances insatisfaisables résolues augmente moins rapidement que le nombre d'instances satisfaisables car il y a beaucoup moins d'instances insatisfaisables (33 *Versus* 53). La table 7.1 récapitule le nombres d'instances résolues suivant le nombre de cœurs utilisés.

#cœurs	SAT	UNSAT	Total
1	15	11	26
2	25	15	40
4	29	23	52
8	31	25	56
16	36	27	63
32	42	28	70

TABLE 7.1 – Résultats de SYRUP sur 1 à 32 cœurs de calculs.

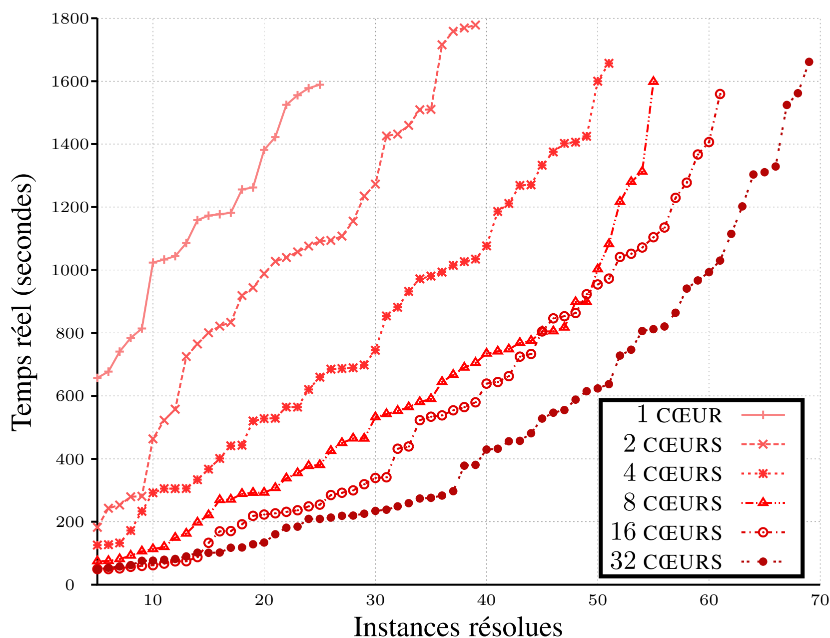


FIGURE 7.1 – Résultats du solveur parallèle SYRUP sur 1 à 32 cœurs de calculs sur les 100 instances du *parallel track* de la SAT Race 2015.

7.1.2 Extensibilité

Dans la littérature, il existe plusieurs manières de calculer les accélérations afin d'obtenir l'extensibilité d'un solveur parallèle. Pourtant, une seule formule mathématique définie théoriquement le *speedup* : pour une instance donnée, son *speedup* est obtenu en divisant le temps réel séquentiel par le temps de la version parallèle sur x cœurs (définition 3.18). Le problème est que, par définition, le *speedup* n'est pas défini pour les instances non résolues séquentiellement. Ainsi, une solution est d'attribuer arbitrairement une valeur comme temps séquentiel sur ces instances. Les manières de choisir cette valeur diffèrent grandement dans la littérature et ont un impact considérable sur l'extensibilité obtenue. Dans la figure 7.2, nous présentons l'extensibilité du solveur SYRUP sur les instances résolues séquentiellement. Nous pouvons observer qu'il est très difficile de faire mieux sur les instances déjà résolues séquentiellement. Cela est prédictible par les accélérations théoriques. En effet, la taille des problèmes déjà résolus n'augmente pas (loi de Gustafson) mais la quantité de code non parallèle augmente avec le nombre de cœurs (Loi d'Amdahl). Néanmoins, plusieurs questions se posent au niveau de l'efficacité des clauses apprises. L'espace de recherche théorique de chaque cœurs se restreint, et l'échange des clauses apprises est censé

éviter le parcours redondant d'un même espace de recherche. Nous pouvons donc attribuer ces accélérations stagnantes à plusieurs facteurs :

- un pipeline surchargé (section 3.2.3, page 64) ;
- les coûts de communications (section 3.5.2, page 78) ;
- un nombre de clauses apprises à gérer de plus en plus élevé.

Principalement, éliminer ou garder trop de clauses apprises réduit l'accélération. Un solveur SAT parallèle *portfolio* partageant les clauses apprises est ainsi potentiellement très sensible aux heuristiques de partage des clauses apprises (chapitre 4).

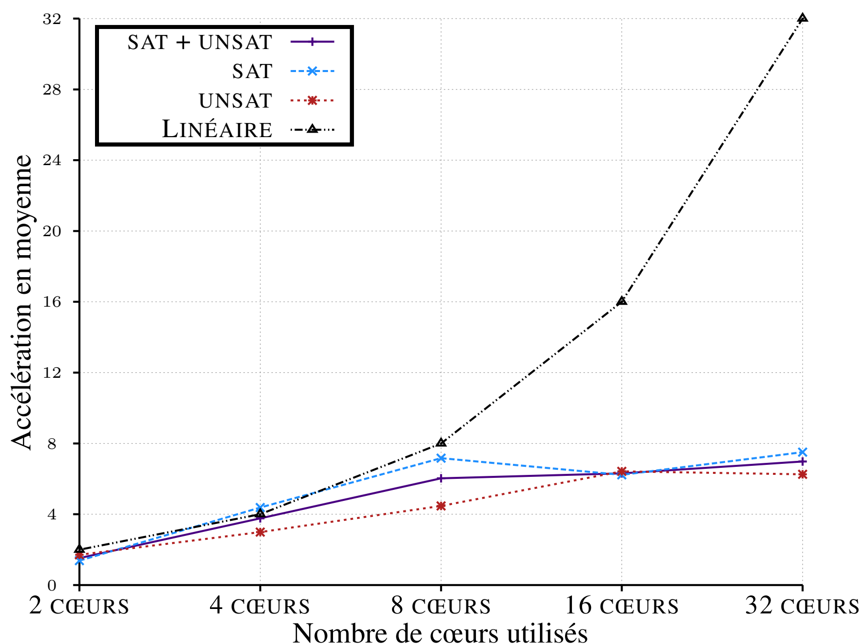


FIGURE 7.2 – Extensibilité de SYRUP de 1 à 32 cœurs sur les instances résolues séquentiellement par la version séquentielle GLUCOSE parmi les 100 instances du *parallel track* de la SAT Race 2015. Chaque point représente l'accélération en moyenne.

Pour pallier ce problème de *speedup*, nous proposons de fixer le temps séquentiel d'une instance non résolue à son temps limite : c'est le temps réel de calcul utilisé afin de résoudre l'instance. En d'autres mots, quand une instance est indéterminée, nous considérons qu'elle est quand même résolue à la fin de cette limite de temps pour le calcul de son *speedup*. Nous appelons cela le *speedup* dans « le pire des cas ». En effet, nous considérons que dans le calcul du *speedup* d'une instance non résolue, celle-ci aurait pu être résolue dans le pire des cas, juste un petit laps de temps après la fin de sa résolution.

Exemple 7.1. Soit un temps limite de 1200 secondes. Nous utilisons dans cet exemple la manière de calculer le *speedup* dans le pire des cas. Si une instance non résolue séquentiellement a été résolue en 1200 secondes avec 2 cœurs, son *speedup* est :

$$S(2) = \frac{T_s}{T_p(2)} = \frac{1200}{1200} = 1$$

Le *speedup* obtenu est donc sous-linéaire car il est strictement inférieur au nombre de cœurs. En effet, dans la pire des situations, l'instance aurait pu être résolue séquentiellement, par exemple, en $1200,001$ secondes.

Nous appelons cela **l'accélération minimum** car elle garantie ainsi qu'elle est la plus petite accélération possible pour une instance indéterminée séquentiellement. De plus, plus nous augmentons le temps de calcul disponible pour la résolution, plus l'accélération minimum devient précise. Notons d'ailleurs que les auteurs du solveur HORDESAT (Balyo *et al.* 2015) augmentent le temps limite de résolution de la version séquentielle (de 1000 secondes à 50000 secondes) afin d'ajouter de la précision dans cette méthode. Ils obtiennent ainsi quelques accélérations super-linéaires, mais aussi d'autres sous-linéaires.

La figure 7.3 expose l'extensibilité de SYRUP obtenue suivant le critère de l'accélération minimum. Nous pouvons aussi observer certaines accélérations super-linéaires notamment quand nous utilisons 8 cœurs de calcul. Néanmoins, nous observons une fois de plus une extensibilité stagnante quand nous augmentons de 8 à 32 le nombre de cœurs. De plus, nous pouvons remarquer que les instances satisfaisables possèdent une meilleure extensibilité que celles insatisfaisables.

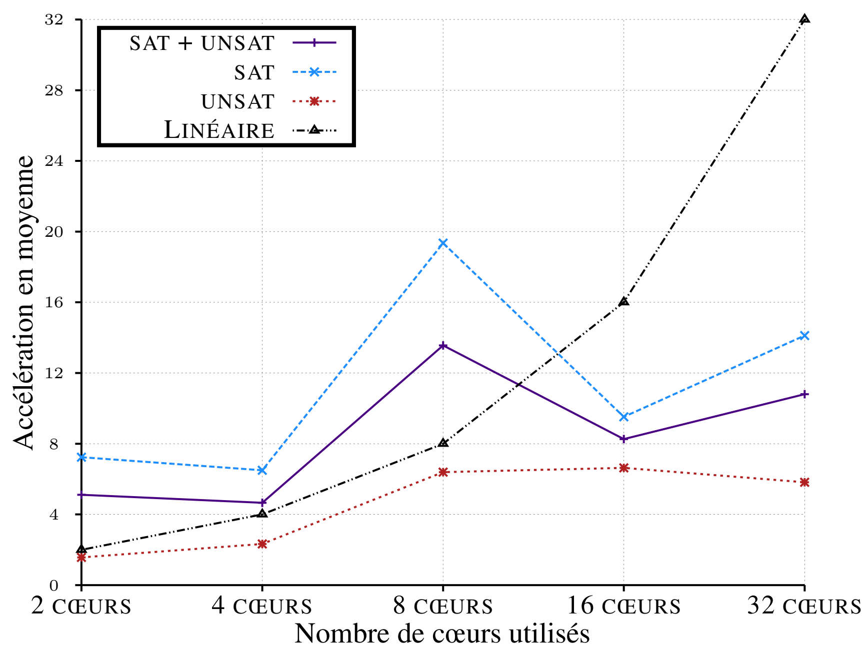


FIGURE 7.3 – Extensibilité de SYRUP de 1 à 32 cœurs sur les toutes instances du *parallel track* de la SAT Race 2015. Chaque point représente l'accélération minimum en moyenne.

Nous avons remarqué que l'extensibilité de SYRUP est moins élevée que celle du solveur HORDESAT (Balyo *et al.* 2015). Pourtant, cela ne signifie pas que HORDESAT est plus extensible. En effet, les temps accordés à la résolution ne sont pas les mêmes et les instances considérées non plus.

Le premier point laisse à penser que lorsqu'une instance est indéterminée séquentiellement, celle-ci a peu de chance d'être résolue en augmentant la limite de temps accordée à sa résolution. Plus précisément, une instance non résolue en x secondes a peu de chance d'être résolue en $x + 1000$ secondes. Par conséquent, l'accélération minimum se retrouve plus élevée quand la limite de temps accordée à la résolution d'une instance est augmentée car cette accélération dépend directement de cette limite. C'est pour cela que le solveur HORDESAT a fixé un *timeout* de 1000 à 50000 secondes. Cela lui permet alors d'avoir des accélérations minimums plus précises.

Le deuxième point insiste sur le fait que les instances de la compétition de 2015 du *parallel track* ont été sélectionnées comme étant les plus dures du *sequential track*.

Les extensibilités produites par les instances varient considérablement et l'indéterminisme joue alors un rôle très important dans ces variations. Par exemple l'instance « vmpc_33.cnf » induit les accéléra-

tions suivantes pour respectivement 2,4,8,16 et 32 cœurs : $\{5.43, 7.34, 63.14, 16.70, 12.53\}$. Ces chiffres exposent alors un gain plus que super-linéaire avec 8 cœurs. La probabilité d'avoir une telle efficacité n'est pas rare sur les instances satisfaisables. En effet, ce phénomène peut être observé sur 15 instances sur un total de 42 instances satisfaisables. Celles-ci comportent des solutions pouvant parfois être plus accessibles via un parallélisme. En effet, ce sont des instances satisfaisables difficiles qui ont beaucoup de variables et très peu de modèles. Donc ces instances peuvent être résolues de temps en temps grâce à de la chance (c'est-à-dire, en retombant sur les bonnes variables directement) et le parallélisme augmente cette probabilité.

Remarquons aussi que sur cette instance, la version de 16 et 32 cœurs a moins bénéficié de « chance ». Par conséquent, celle-ci expose alors une extensibilité décroissante. Dans le cas contraire, nous pouvons observer quelques accélérations constantes, voir décroissantes, sur des instances très difficiles. À titre d'exemple, l'instance « 008-80-12.cnf » est résolue un petit peu plus rapidement en parallèle. En effet, ses accélérations sont $\{.95, 1.68, 1.42, 1.13, 1.63\}$.

La prochaine étape est alors d'utiliser plus de cœurs de calcul tout en essayant d'avoir une bonne extensibilité. Néanmoins, les machines possédant un nombre de cœurs supérieur à 32 sont très chères et restent toujours limitées à un nombre donné de cœurs. Afin de dépasser cette limite, une autre manière d'obtenir plus de cœurs consiste à considérer plusieurs ordinateurs comme ceux disponibles via les *clusters* de calcul ou plus récemment via le *cloud computing*. Malheureusement, SYRUP a été conçu pour fonctionner uniquement sur une architecture *multi-thread* (à mémoire partagée). Comme une architecture distribuée est appropriée pour utiliser plus de cœurs disponibles, nous devons alors programmer une version de SYRUP distribuée. Nous appelons alors cette dernière D-SYRUP.

Un autre point important concerne la manière de communiquer les informations entre les solveurs. Quand nous utilisons SYRUP sur une architecture *multi-thread*, les informations sont échangées en utilisant la mémoire partagée presque gratuitement en terme de temps de calcul. Néanmoins, quand nous considérons une architecture distribuée composée de plusieurs ordinateurs, les communications peuvent être coûteuses et ne s'implémentent pas si facilement. Ce chapitre expose alors les contributions consacrées à cette fin. Dans les prochaines sections, nous réalisons une étude poussée des modèles de programmation dans le cadre d'un solveur SAT parallèle.

7.2 L'architecture distribuée

Les solveurs SAT parallèles partagent une énorme quantité d'informations. Lorsque nous considérons l'architecture à mémoire partagée, ces informations peuvent être efficacement collectées par les différents solveurs en utilisant cette dernière. Comme tous les processus d'un ordinateur partagent un seul espace d'adresses de données, les communications entre les solveurs peuvent être aussi rapide que les accès à la mémoire, même si en pratique, certaines copies sont réalisées afin d'accroître les performances du solveur. Deux méthodes peuvent être employées en considérant le partage d'informations dans une architecture distribuée : celle centralisée et celle décentralisée. Ces deux méthodes sont décrites dans la section 3.6.2, définitions 3.32 et 3.33. Pour rappel, la première manière dite centralisée échange les informations en les envoyant à un processus maître qui les renvoi aux autres unités de calcul. En revanche, la seconde dite décentralisée envoie directement les informations aux autres unités.

7.2.1 AMPHAROS

La conclusion du chapitre 6 (section 6.5, page 151) expose un problème dans le solveur AMPHAROS. Pour rappel, grâce à AMPHAROS, nous avons expérimentalement démontré qu'une manière centralisée

en charge de collecter et partager les informations entraîne une congestion rapide du réseau. Ce phénomène induit alors une baisse significative des performances. Plus précisément, l'échange des clauses apprises n'apporte alors plus aucun gain sur la plupart des problèmes. En addition à ce problème, plus le nombre de cœurs de calcul augmente, plus le coût des communications est important. Par conséquent, partager d'une manière centralisée les clauses n'est potentiellement pas la bonne solution car l'objectif d'une architecture distribuée dans un solveur SAT parallèle est d'utiliser un nombre considérable d'ordinateurs.

7.2.2 Interblocages

Afin de pallier le problème du goulot d'étranglement induit par l'architecture centralisée, dans la suite de ce chapitre, nous utilisons une architecture décentralisée. En d'autres mots, dans celle-ci et contrairement à l'architecture centralisée, chacun des processus communique directement les informations avec tous les autres, sans passer par un processus maître. Cependant, notons qu'un bouchon dans les communications peut encore survenir dans cette architecture décentralisée quand le nombre de clauses à partager est trop élevé. De plus, les communications réalisées dans cette architecture doivent être réalisées prudemment afin d'éviter les *deadlocks* (interblocages en français). Notons bien ici que nous parlons des *deadlocks* engendrés par l'échange de messages via le réseau, pas ceux induit par la mémoire partagée et leurs sections critiques associées (*mutex*). Par ailleurs, une explication de ces deux sortes de *deadlocks* est réalisée dans l'état de l'art, section 3.6.2, page 86. Un *deadlock* apparaît quand un processus demande une ressource qui est déjà occupée par d'autres processus. Par exemple, considérons le cas ou chaque processus est implémenté tel que qu'il réalise respectivement une opération `send` bloquante puis une opération `receive` elle aussi bloquante. Le terme « bloquante » signifie alors ici que l'opération `send` (resp. `receive`) est bloquée tant que le message n'a pas été envoyé (resp. reçu) sur le réseau. Dans une telle situation, si deux processus envoient chacun un message à l'autre, alors chaque processus doit attendre que le message soit reçu par l'autre pour continuer. Par conséquent, ils sont tous les deux bloqués dans leurs opérations `send` et ce phénomène est alors appelé un *deadlock*. Plus précisément, une opération `send` n'est pas complétée tant que l'opération `receive` correspondante n'est pas exécutée.

7.2.3 Les cycles de communications

Ce problème est bien connu et plusieurs solutions existent afin d'éviter et/ou détecter ces *deadlocks* (Kshemkalyani et Singhal 1994, Balaji *et al.* 2010). Au niveau des solveurs SAT parallèles, la plupart des approches proposées dans la littérature (Ehlers *et al.* 2014, Balyo *et al.* 2015, Audemard *et al.* 2016a) utilisent des communications non-bloquantes et/ou collectives pour éviter ce problème. Généralement, la solution retenue consiste à attendre que tous les $\text{BUFFERS}_{network}$ contenant les messages à envoyer peuvent être consultés et modifiés sans risque de *deadlocks*. Cela consiste à attendre que les messages ont été envoyés (les messages n'ont pas été nécessairement reçues). Cette solution est réalisé via ce que nous appelons un cycle de communications. C'est une boucle effectuant plusieurs tâches qui est répétée tant que la solution n'a pas été trouvée :

- Exécution de toutes les communications des données d'un $\text{BUFFER}_{network}$ (`MPI_SEND` et `MPI_RECEIVE`);
- Attente des terminaisons des communications (`MPI_WAIT_ALL`);
- Attente de 5 secondes. Durant ce temps, un *thread* solveur remplit le $\text{BUFFER}_{network}$ de clauses.

Nous étudions cette approche pour la version distribuée de SYRUP présentée dans ce chapitre.

7.2.4 Phase de recherche et phase de communications

Un autre point important concerne la manière de réaliser les communications entre les solveurs. Deux solutions sont généralement considérées :

- Un *thread* alterne périodiquement étapes de recherches et de communications (c'est la solution retenue par le solveur AMPHAROS présenté dans nos contributions du chapitre 6) ;
- Deux *threads* distincts réalisent séparément ces deux tâches (c'est la solution retenue par les auteurs de HORDESAT Balyo *et al.* (2015) et TOPOSAT Ehlers *et al.* (2014)).

Dans la suite de ces travaux, nous décidons d'associer des *threads* exclusivement dévoués à l'échange d'informations (clauses apprises et notification de la solution) et d'autres exclusivement dédiés à la recherche afin de résoudre le problème. Dans une telle configuration, chaque *thread* communicateur exécute des cycles de communications. Rappelons qu'un cycle est composé principalement de deux phases : l'une partageant les clauses et l'autre est une phase d'attente nécessaire au remplissage des $\text{BUFFERS}_{network}$. Durant un cycle de communications et afin d'éviter les *deadlocks*, le *thread* communicateur utilise une section critique (*mutex*) et doit attendre que les $\text{BUFFER}_{network}$ d'envoi soient à deux reprises disponibles (via `MPI_WAIT_ALL`) : une première fois pour vérifier si une solution a été trouvée puis une seconde fois pour échanger les clauses apprises.

7.2.5 Objectifs

Dans le reste de ce chapitre, nous évaluons empiriquement une version distribuée du solveur *multi-thread* SYRUP en utilisant les *threads* communicateurs mentionnés précédemment. Dans cette évaluation, nous faisons la distinction entre ce que nous nommons le modèle de programmation pur par passage de messages, ou les couples de deux *threads* (solveur, communicateur) sont considérés comme fonctionnant sur des ordinateurs indépendants (permet d'utiliser uniquement une librairie MPI) et le modèle de programmation partiellement hybride, ou les communications entre les solveurs sur la même machine sont réalisées en utilisant la mémoire partagée (utilisant à la fois une librairie MPI et une autre pour la mémoire partagée). Une présentation des modèles de programmations hybrides est réalisée dans l'état de l'art (section 3.6.2, page 86).

7.3 Le modèle de programmation pur par passage de messages

Ce modèle de programmation respecte le protocole de communication présenté dans la section précédente. En plus de cela, il se définit via certaines contraintes :

- Il utilise plusieurs processus par ordinateur ;
- Chaque solveur SAT séquentiel forme un processus : il est capable d'effectuer la recherche aussi bien que les communications grâce à l'aide d'un ou de deux *threads* (solveur et communicateur).
- Les données sont répliquées pour chaque solveur à la fois sur le réseau et dans la mémoire partagée.

La figure 7.4 illustre ce modèle de programmation avec deux *threads* par processus solveur. Le processus solveur PROCESSUS_1 représenté par un carré en pointillé, regroupe un *thread* de recherche S_1 (un solveur) et son propre *thread* communicateur C_1. Les données (les clauses apprises) son partagées entre ces deux *threads* en utilisant la mémoire partagée via un tampon de données nommée BUFFER dans la figure 7.4 et dans la suite de ce chapitre.

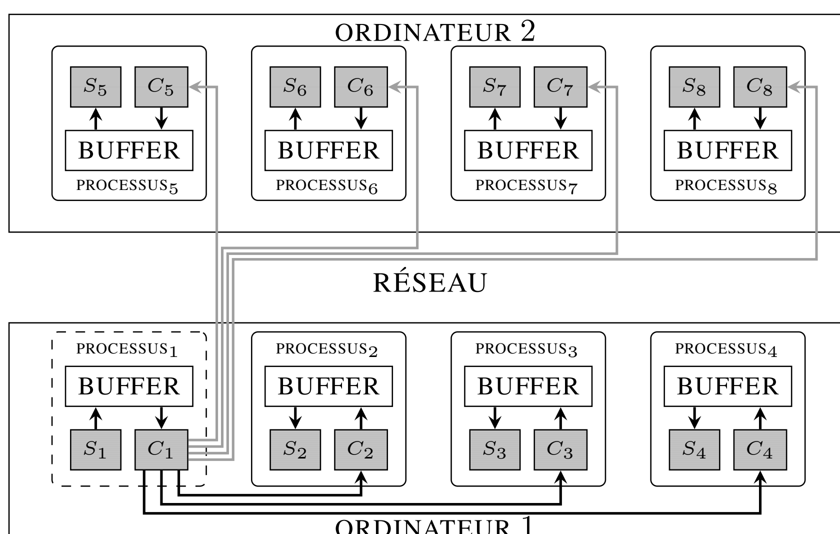


FIGURE 7.4 – Le modèle de programmation pur par passage de messages : l'accent est réalisé sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs sur une configuration possédant deux ordinateur.

Sur cette figure, nous nous focalisons sur les données envoyées par le *thread* de recherche S_1 à la totalité des autres solveurs représentés par les autres *threads* de recherche S_n via les processus PROCESSUS$_n$ avec $n \neq 1$. Pour cela, les clauses apprises sont d'abord transférées du *thread* S_1 à son *thread* communicateur C_1 via un BUFFER utilisant la mémoire partagée (bibliothèque *multi-thread*). Ensuite, chaque *thread* communicateur C_n tel que $n \neq 1$ reçoit les clauses apprises à partir de C_1 uniquement par passage de messages (via une bibliothèque réseau) même quand deux *threads* sont sur la même machine. Pour finir, chaque *thread* communicateur C_n envoie les clauses apprises reçues à leur *thread* de recherche respectif S_n (associé au PROCESSUS$_n$) tel que $n \neq 1$.

7.3.1 Désavantages

En réalité, dans ce modèle de programmation pur par passage de messages, chaque solveur se comporte comme un ordinateur indépendant. Il est très important de noter que dans cette situation, même si une bibliothèque de la mémoire partagée est utilisée pour échanger les clauses entre un *thread* de recherche et un *thread* communicateur, elle n'est pas utilisée pour échanger ces clauses entre les *threads* communicateurs (ou même encore de recherche). Ainsi, le principal défaut de ce modèle est la réplication des données. Plus particulièrement, celle-ci induit une diminution notable de la vitesse des communications.

Plus précisément, sur un même ordinateur, cette manière de fonctionner oblige la copie de données entre les solveurs sans utiliser l'espace d'adressage global habituellement induit par la mémoire partagée (flèche noire). Au lieu de cela, seul le passage de messages est utilisé entre deux processus, peu importe si ils sont sur la même machine ou pas. De plus, quand les données doivent être envoyées aux solveurs situés sur une autre machine, ce modèle exige d'envoyer un message à chaque solveur séquentiel (et donc à chaque processus) plutôt qu'un seul par machine (flèche grise). À titre d'exemple, dans la figure 7.4, lorsque le communicateur C_1 envoie une clause à la deuxième machine, celle-ci est envoyée 4 fois plutôt qu'une seule fois. De ce fait, dans notre exemple, le réseau est 4 fois plus encombré. Par conséquent, la même donnée est répliquée et envoyée sur le réseau autant de fois que le nombre de solveurs inclus dans la machine les réceptionnant, et cela conduit généralement à la congestion du réseau.

7.3.2 Avantages

Malgré ces défauts, l'avantage de ce modèle de programmation est la possibilité d'utiliser uniquement une librairie de passage de messages tel que MPI (voir section 3.6.2) ou même uniquement la programmation par *socket*. Cela peut faciliter grandement le développement et la correction de bogues. Ceci signifie qu'en utilisant ce modèle de programmation, un solveur distribué peut directement être créé en étendant un solveur SAT séquentiel plutôt qu'un *multi-core*.

7.3.3 Utilisations

Le modèle de programmation pur par passage de messages est la solution retenue par les auteurs de TOPOSAT (section 4.1.10) afin de réaliser leurs communications. Ils utilisent des cycles (voir section 4.1.10) de communications de 5 secondes. Un point clé de l'architecture de TOPOSAT est l'utilisation de communications point à point et non bloquantes afin de construire différentes topologies d'échange de messages (*2-dimensional grid*, *medium-coupled*, ...). Ces topologies visent alors à réduire la congestion du réseau induit par ce modèle de programmation. Toutefois, notons que nos objectifs ne sont pas, dans ce chapitre, de comparer les topologies. De ce fait, nous implémentons pour chaque modèle présenté dans ce chapitre, la topologie où chaque processus communique avec tous les autres, autrement dit, celle sans aucune restriction.

Ce modèle est aussi la solution que nous avons employé dans le solveur « diviser pour mieux régner » AMPHAROS (partie contribution, chapitre 6). Contrairement à ce qui vient d'être présenté dans cette section, AMPHAROS a la particularité de n'employer qu'un seul *thread* par processus. Ce *thread* alors similaire à un processus qui effectue à la fois les phases de recherche et de communication. Cette manière de faire ne retire aucun des défauts de ce modèle de programmation. Nous montrons alors expérimentalement (section 6.4.3) une baisse des performances anormales lors de l'échange des clauses apprises dans ce solveur et ces travaux ont alors pour objectif de régler ce problème.

7.3.4 Expérimentations

Afin de comparer différents modèles de programmation, nous avons développé un modèle pur par passage de messages basé sur le solveur séquentiel GLUCOSE. Cette implémentation fournit directement une version distribuée de SYRUP en utilisant les *threads* communicateurs présentés dans cette section.

Premièrement, nous avons étudié l'impact de la taille des BUFFER transférant les clauses entre un *thread* de recherche S_n et un *thread* communicateur C_n, ainsi que la fréquence des échanges (via les cycles de communications). Afin d'examiner les performances globales du solveur pendant cette étude, nous exposons le nombre de propagations unitaires par seconde. Pour réaliser cette expérimentation, nous exécutons 256 processus sur 32 ordinateurs composés chacun de 8 cœurs durant 100 secondes (temps réel). Nous avons sélectionné 20 instances provenant de la compétition SAT Race 2015 (*parallel track*) et non résolues par le solveur SYRUP en 300 secondes. Ces résultats ont été obtenus dans la section 7.1 avec 32 cœurs de calcul. Il représente un ensemble de familles d'instances variées et difficilement résolvables par SYRUP. Par conséquent, ces 20 instances peuvent être vues comme un échantillon représentant diverses instances difficiles. Afin d'évaluer l'impact de la taille du BUFFER et les différents temps de cycles de communications sur l'efficacité du solveur, trois tailles ont été sélectionnées pour les BUFFER (20MB, 100MB, et 200MB) et trois différents cycles de communications ont été considérés (tous les 0.5, 5, et 10 secondes).

Le tableau 7.2 reporte pour chaque combinaison de taille du BUFFER et de temps de cycle de communications, la moyenne du nombre de clauses reçues noté « Reçues » dans la table, la moyenne du

nombre de clauses réellement retenues par les solveurs noté « Retenues » (ce sont les clauses importées à travers le réseau et qui ont traversées la mémoire partagée (le BUFFER) et sont alors *1-watched* : certaines peuvent être supprimées suivant la taille du BUFFER), la moyenne du nombre de clauses retenues qui entrent en conflit (ces clauses passent de *1-watched* à *2-watched*, voir section 4.1.8, page 98) notée « Ret. (conflits) », la moyenne du nombre de clauses retenues utilisées durant le processus de la propagation unitaire notée « Ret. (utiles) », la moyenne du nombre de propagation unitaire par seconde notée « Propagation » et, pour finir, la moyenne du nombre de conflits par seconde notée « Conflits ». Ces mesures sont effectuées sur toutes les instances pendant leur temps réel d'exécution de 100 secondes sur 32 machines pour un total de 256 cœurs de calcul (32 *bi-processors Intel XEON X7550*). Afin de ne pas ralentir le solveur, elles sont récupérées uniquement toutes les 5 secondes pendant la recherche. Pour chaque mesure, une moyenne globale est calculée à partir des données apportées par les 20 instances.

	20MB			**100MB**			**200MB**		
Com. Cycle	**0.5s**	**5s**	**10s**	**0.5s**	**5s**	**10s**	**0.5s**	**5s**	**10s**
Reçues	8,270	30,989	38,820	9,920	28,369	39,890	11,007	28,296	34,346
Retenues	82%	25%	0.8%	93%	51%	34%	93%	71%	62%
Ret. (conflits)	43	55	26	54	51	16	72	33	12
Ret. (utiles)	494	618	306	553	575	173	783	452	192
Propagations	100,379	489,289	525,738	91,103	479,145	540,286	69,912	469,021	443,696
Conflits	216	823	1,101	254	734	1,069	275	721	891

TABLE 7.2 – Modèle de programmation pur par passage de messages. Impact de la taille des BUFFER et de la fréquence des échanges (cycles de communications) sur les performances globales des solveurs. Tous ces résultats correspondent à la moyenne obtenue par tous les processus par seconde.

Impact du temps des cycles de communication

Premièrement, nous pouvons observer que quelque soit la taille des BUFFER, les pires résultats sont obtenus pour les cycles de communication de 0.5 secondes. En effet, les nombres de propagations et de conflits par seconde sont considérablement affectés : 5 fois moins qu'en utilisant des cycles de 5 ou 10 secondes. Cela est causé par les opérations bloquantes exécutées durant un cycle de communication (`MPI_WAIT_ALL` et opérations utilisant les *mutexs*). Car plus les cycles sont courts, plus ses opérations sont réalisées fréquemment. De ce fait, dès que nous augmentons le temps des cycles de communications à 5 secondes, le nombre d'appels à ces opérations bloquantes est divisé par 10. Par conséquent, les performances globales du solveur en terme de propagations et de conflits sont meilleures.

Deuxièmement, nous pouvons aussi remarquer que le nombre de clauses retenues diminue quand le temps d'un cycle de communication augmente. En effet, dans ce cas, plus de clauses doivent être partagées en un cycle. Par conséquent, le BUFFER de la mémoire partagée a plus de chance d'être plein. Dans ce derniers cas, les clauses ne sont simplement pas partagées et sont alors supprimées du BUFFER (voir la section 4.1.8 pour plus de précision sur la gestion de ce BUFFER dans SYRUP). Cela est un potentiel défaut, car certaines de ces clauses non partagées peuvent être utiles au solveur.

Impact de la taille des BUFFER

À présent, nous nous focalisons sur un temps des cycles de communication de 5 secondes. Nous observons que plus nous augmentons la taille des BUFFER, plus le pourcentage de clauses retenues est

élevé. Effectivement, la même quantité de données est partagée en un cycle, mais moins de clauses sont supprimées. Cela peut être vu comme un bon point car augmenter la taille du BUFFER permet donc d'apporter plus de clauses aux solveurs S_n. Néanmoins, le nombre de clauses retenues étant utiles et entrant en conflit, aussi bien que le nombre de propagations et de conflits diminuent légèrement quand la taille du BUFFER augmente. Cela est expliquer par une surcharge des solveurs à cause de la gestion de la base de clauses apprises. Plus précisément, un nombre trop élevé de clauses apprises ralentit les solveurs séquentiels car il y a de plus en plus de littéraux à examiner durant la propagation unitaire.

Synthèse

Afin d'obtenir les meilleures performances, nous avons fixé ces deux paramètres :

- le temps d'un cycle de communication à 5 secondes ;
- taille du BUFFER de la mémoire partagée à 20MB.

7.4 Le modèle de programmation partiellement hybride

Dans cette section, nous proposons un modèle de programmation partiellement hybride comme une alternative afin de pallier les inconvénients du modèle de programmation pur par passage de messages. Ce modèle de programmation se définie par certaines caractéristiques :

- il intègre un seul processus par ordinateur ;
- plusieurs solveurs SAT séquentiels forment un processus : il s'agit d'un regroupement de plusieurs *threads* de recherche associés à un seul *thread* communicateur ;
- Deux librairies indépendantes sont utilisées pour communiquer : une gère la mémoire partagée tandis qu'une autre gère les communications sur le réseau par passage de messages ;
- Le *thread* communicateur regroupe les données qui doivent être envoyées sur le réseau et distribue celles qu'il reçoit.

Le *thread* communicateur est utilisé comme une interface afin de partager les données entre des solveurs localisés sur des ordinateurs différents tandis que la mémoire partagée permet d'échanger les données entre les solveurs sur le même ordinateur. La figure 7.5 décrit ce schéma. Chaque ordinateur utilise un seul processus qui regroupe tous les solveurs SAT séquentiels d'une machine et un *thread* communicateur.

Cette figure décrit la transmission de clauses du solveur S_1 aux autres solveurs S_n tel que $n \neq 1$. Pour cela, le solveur S_1 envoie d'abord ses clauses apprises aux *threads* solveurs S_2, S_3, S_4 puis à son *thread* communicateur C_1 en utilisant le tampon localisé dans la mémoire partagée (BUFFER). Remarquons que les BUFFER du modèle de programmation pur par passage de messages et de ce modèle sont les mêmes aux niveaux de leurs implémentations, seul leur utilisation change. Par la suite, le communicateur C_1 envoie les clauses à travers le réseau au communicateur C_2. Pour finir, une fois que le communicateur C_2 a récupéré les clauses, il les distribue aux solveurs S_5, S_6, S_7 et S_8.

7.4.1 Avantages et désavantages

De toute évidence, le principal avantage de cette approche est la mutualisation des données échangées à la fois sur le réseau et via la mémoire partagée. L'hybridation permet d'utiliser proprement les deux

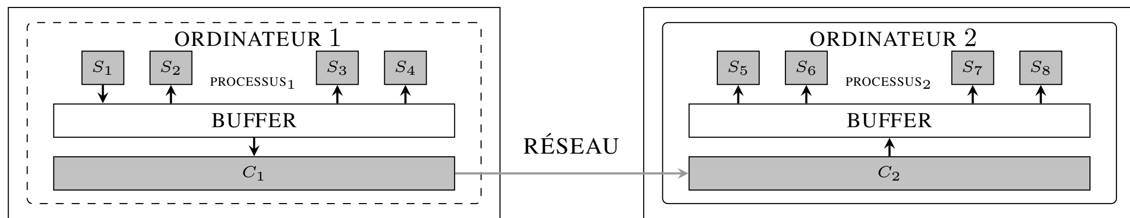


FIGURE 7.5 – Le modèle de programmation partiellement hybride : une focalisation est réalisée sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs avec deux ordinateurs.

sortes d'échange de données en fonction de la localisation des *threads* (sur la même machine ou sur des machines différentes). Ainsi, ce modèle corrige le défaut le plus important du modèle de programmation pur par passage de messages. À titre d'exemple, une clause doit être dupliquée 4 fois en mémoire puis envoyée 4 fois sur le réseau afin d'atteindre tous les processus d'une autre machine dans la figure 7.4 représentant une possible configuration du modèle de programmation pur par passage de messages. Au lieu de cela, le modèle partiellement hybride représenté par la figure 7.5 met en mémoire une seule fois la clause et l'envoie aussi qu'une seule fois sur le réseau pour atteindre une autre machine. Par conséquent, les données échangées à travers le réseau sont considérablement réduites.

Plus généralement, ce gain peut être exprimé en nombre de messages $m(x, o)$ nécessaire afin de transmette une clause en fonction du nombre de solveurs SAT séquentiels par ordinateur noté x et du nombre d'ordinateurs o. Remarquons que ce nombre x est la plupart du temps égal aux nombres de cœurs de calcul d'une machine. Nous supposons donc que toutes les machines possèdent le même nombre de cœurs et de solveurs. Nous avons :

$$m(x, o)_{pur} = x \times o \text{ et } m(x, o)_{hybride} = x$$

Toutefois, ce modèle de programmation possède un léger défaut. L'obligation d'utiliser deux bibliothèques différentes induit un code beaucoup plus volumineux, moins lisible et donc potentiellement sujet à un plus grand nombre de bogues. De plus, les bibliothèques considérées pour faire une telle hybridation ont leur importance. Certaines bibliothèques réseaux ne prennent pas en compte le *multi-thread*. Cela peut induire quelques problèmes notamment aux niveaux des *deadlocks*. Nous en parlons dans la prochaine section.

7.4.2 Utilisation

Dans les cycles de communication, les clauses sont mises dans un tampon noté $\text{BUFFER}_{network}$ avant leur envoi et dès leur réception. En effet, c'est le même tampon qui sert à envoyer les clauses et à les recevoir via le réseau. Il est important de noter que ce tampon $\text{BUFFER}_{network}$ n'est pas le même que celui employé depuis le début de ce chapitre pour la mémoire partagée BUFFER. Plus précisément, rappelons que BUFFER est utilisé pour échanger les clauses entre plusieurs *threads* de recherche sur la même machine. En revanche, $\text{BUFFER}_{network}$ est dédié à partager les clauses entre différentes machines via le réseau.

Le solveur HORDESAT est basé sur ce modèle de programmation. Sa principale caractéristique est la gestion des messages envoyés via le *thread* communicateur. Il utilise des cycles de communication de 5 secondes. Sa particularité est d'avoir une communication collective des données d'une taille fixe toutes les 5 secondes. Plus précisément, les tampons $\text{BUFFER}_{network}$ de chaque *thread* communicateur possède la même taille fixée à 1500 entiers. Autrement dit, quand HORDESAT est exécuté sur o machines,

sur une machine donnée, 1500 entiers de données son envoyés aux autre machines et cette machine reçoit $1500 \times (o-1)$ entiers à chaque cycle de communication.

La taille fixe du BUFFER$_{network}$ d'HORDESAT induit quelques problèmes. Si le nombre de clauses ne dépasse pas les 1500 entiers au moment de l'envoi du message sur le réseau, un rembourrage de 0 est ajouté au BUFFER$_{network}$ afin d'obtenir la taille désirée. À l'inverse, lorsque la limite de taille est atteinte avant l'envoi, les clauses ne sont simplement pas ajoutées au BUFFER$_{network}$.

Afin d'éviter ces problèmes, les auteurs d'HORDESAT (Balyo *et al.* 2015) proposent d'adapter les heuristiques des solveurs SAT séquentiels afin qu'ils produisent plus ou moins de clauses en fonction de la demande du BUFFER$_{network}$. Ainsi, pour un cycle de communication, si le nombre de clauses n'est pas suffisant afin de remplir totalement le BUFFER$_{network}$, alors une fonction est appelée afin d'encourager heuristiquement les solveurs SAT séquentiels à produire plus de clauses pour le prochain cycle de communication. Inversement, quand le BUFFER$_{network}$ a atteint sa limite et ne peut plus accueillir de clauses, les solveurs SAT séquentiels sont amenés à produire moins de clauses apprises pour le prochain cycle. Notons aussi qu'avant de mettre les clauses dans le BUFFER$_{network}$, celles-ci sont triées d'une manière heuristique (suivant leurs tailles et leurs valeurs LBD) afin de transmettre les « meilleures » en priorité.

7.4.3 Implémentation

Nous avons développé notre propre solveur partiellement hybride, en utilisant les *threads* communicateurs présentés au début de cette section. Ce solveur est basé sur le solveur parallèle *multi-thread* SYRUP (version 4.1). Rappelons que dans ce modèle de programmation, nous devons ajuster deux tampons différents : celui utilisé pour partager les clauses entre les différents *threads* de recherche sur une même machine et utilisé par un appel du solveur SYRUP, et le tampon nécessaire à l'envoi et la réception des clauses sur le réseau. Il y a un BUFFER$_{network}$ par machine, c'est-à-dire, un seul par *thread* communicateur. Néanmoins, nous pensons que les communications collectives de taille fixe à chaque cycle faites par HORDESAT pour envoyer les clauses sur le réseau sont un obstacle à l'efficacité des solveurs SAT distribués. Ainsi, nous proposons d'effectuer une communication collective qui échange des données de tailles différentes. En d'autres mots, nous ne limitons pas la taille des BUFFER$_{network}$. De ce fait, lors d'un cycle, une machine peut alors envoyer directement un très grand nombre de clauses tandis qu'une autre peut en envoyer un très petit nombre. Notre objectif est donc opposé à HORDESAT, puisque nous avons choisi de ne pas modifier la manière dont les clauses sont produites par les solveurs SAT séquentiels. De plus, cela évite d'envoyer inutilement des données composées de zéro sur le réseau et de ne pas écarter quelques clauses qui auraient dû être partagées. En pratique, cela est permis par les bibliothèques MPI ou la programmation par *socket*. Plus précisément, HORDESAT fait appel à la fonction `MPI_AllGATHER` tandis que notre solveur utilise la fonction `MPI_AllGATHERV`.

7.4.4 Expérimentations

Comme pour la section précédente 7.3 (le modèle de programmation pur par passage de messages), nous étudions également l'impact de la taille du BUFFER et la fréquence des échanges (temps des cycles de communication) sur les performances globales des solveurs pour ce modèle avec les mêmes instances. Les résultats sont présentés dans le tableau 7.3. Ils montrent la même tendance que pour le modèle de programmation pur par passage de messages (tableau 7.2). En effet, nous pouvons également observer que des cycles de communication courts pénalisent les performances des solveurs SAT séquentiels et qu'une grande taille du tampon BUFFER amène à réduire le nombre de clause utiles retenues.

	20MB			100MB			200MB		
Com. Cycle	**0.5s**	**5s**	**10s**	**0.5s**	**5s**	**10s**	**0.5s**	**5s**	**10s**
Reçues	28,202	38,448	41,826	26,980	36,505	36,370	26,836	34,643	35,427
Retenues	84%	68%	43%	86%	85%	81%	86%	86%	86%
Ret. (conflits)	342	318	209	325	280	190	326	258	176
Ret. (utiles)	3,841	3,256	2,578	3,429	2,688	2,008	3,647	2,603	1,853
Propagations	504,540	636,850	658,177	528,529	577,653	603,685	511,709	559,477	595,084
Conflits	976	1,364	1,283	928	1,253	1,248	929	1,184	1,203

TABLE 7.3 – Modèle de programmation partiellement hybride. Impact de la taille des BUFFER et de la fréquence des échanges (cycles de communications) sur les performances globales des solveurs. Tous ces résultats correspondent à la moyenne obtenue par tous les processus par seconde.

Un point important est que le modèle partiellement hybride fournit des taux de propagations et de conflits par seconde plus élevés que le modèle pur par passage de messages. Plus précisément, le taux de propagations (resp. de conflits) par seconde est entre 69,912 et 525,738 (resp. 216 et 1,101) pour le pur tandis qu'il est entre 504,540 et 658,177 (resp. 929 et 1,364) pour le modèle partiellement hybride. Cela montre que cette approche semble plus bénéfique. De plus, comme pour le modèle de programmation pur par passage de messages, un BUFFER fixé à 20MB et des cycles de communication de 5 secondes semblent être le meilleur compromis.

7.5 Le modèle de programmation complètement hybride

Comme peut le suggérer les résultats du modèle de programmation partiellement hybride (tableau 7.3), il semble bénéfique de partager les clauses à travers le réseau aussi vite que possible. En effet, nous observons que la quantité de clauses retenues (lignes Ret. (conflit) et Ret. (utiles)) est, quelque soit la taille du BUFFER choisie, plus importante quand la fréquence des cycles de communication augmente, c'est-à-dire, quand le temps des cycles de communication diminue. Néanmoins, quand les cycles de communication sont très court (0,5 secondes), nous pouvons remarquer que le modèle de programmation partiellement hybride souffre d'un problème lié à ses sections critiques (*mutexs* et `MPI_WAIT_ALL`). En effet, ce problème réduit considérablement l'efficacité du solveur. Plus précisément, avec un BUFFER de 20MB, nous avons 504,540 propagations (resp. 976 conflits) par seconde pour des cycles de 0.5 secondes alors que nous en avons 636,850 propagations (resp. 1,364 conflits) par seconde avec des cycles de communication de 5 secondes. Il est important de remarquer que ces chiffres sont respectivement les bornes minimums et maximums des résultats obtenus quelque soit la taille du BUFFER (lignes propagations et conflits du tableau 7.3). Par conséquent, cela démontre que les sections critiques ont un rôle très importants dans l'efficacité d'un solveur SAT parallèle. Par section critique, nous parlons des deux moyens habituellement employés afin d'éviter les problèmes de *deadlock*, à savoir, l'utilisation des *mutexs* avec la mémoire partagée et l'utilisation d'opérations bloquantes comme `MPI_WAIT_ALL` avec le passage de messages. Dans cette section, nous abordons alors ce problème en introduisant un algorithme qui possède deux objectifs orthogonaux :

- réduire le nombre d'accès aux sections critiques ;
- envoyer les clauses aussi vite que possible.

7.5.1 Méthode

Le but est donc d'envoyer les clauses directement sur le réseau, c'est-à-dire, sans attendre un cycle de communication. Pour cela, nous modifions les *threads* de recherche afin qu'il envoient eux-mêmes les clauses sur le réseau. Ainsi, les *threads* de recherche font une partie du travail des *threads* communicateurs : l'envoi des clauses sur le réseau. Par conséquent, le *thread* communicateur d'un processus (de chaque ordinateur) devient un *thread* receveur puisque son seul travail est à présent de recevoir les clauses. Contrairement au modèle de programmation partiellement hybride, ce modèle de programmation permet à plusieurs *threads* de communiquer des données sur le réseau en même temps. Nous appelons donc une telle approche le modèle de programmation complètement hybride. Il est donc définit par les contraintes suivantes :

- il intègre un seul processus par ordinateur ;
- plusieurs solveurs SAT séquentiels forment un processus : il s'agit d'un regroupement de plusieurs *threads* ayant les fonctionnalités de recherche et d'envoi des clauses sur le réseau associés à un seul *thread* les recevant alors nommé receveur ;
- deux librairies indépendantes sont utilisées pour communiquer : une gère la mémoire partagée tandis qu'une autre gère les communications réseau par passage de messages ;
- les *threads* receveurs distribuent les données qu'ils reçoivent à leurs *threads* de recherche.

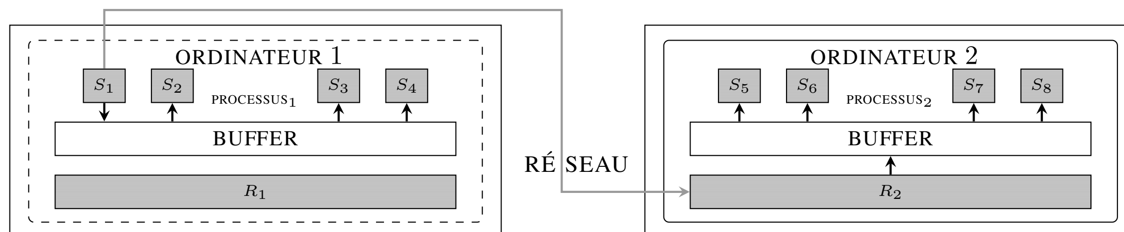


FIGURE 7.6 – Modèle de programmation complètement hybride : Un focalisation est réalisée sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs avec deux ordinateurs.

La figure 7.6 représente le modèle de programmation complètement hybride. Chaque ordinateur possède un seul processus qui intègre tous les *threads* de recherche et un *thread* receveur. Dans cette figure, le solveur S_1 envoie une clause aux solveurs S_2, S_3 et S_4 via la mémoire partagée (BUFFER) et au deuxième ordinateur via le réseau. Par la suite, le *thread* R_2 reçoit cette clause et l'envoie au solveurs S_5, S_6, S_7 et S_8 via la mémoire partagée.

Notons bien que dans notre méthode, il n'y a pas de cycle de communication. À la place, les clauses sont envoyées une par une par les *threads* de recherche via une communication bloquante (`MPI_SEND`). Les *threads* receveurs récupèrent constamment les clauses via une boucle exécutant l'opération bloquante `MPI_RECEIVE` tant que la satisfaisabilité ou l'insastifaisabilité de l'instance n'a pas été trouvée. Lorsque qu'une clause est reçue via un `MPI_RECEIVE`, celle-ci est directement placée dans le (BUFFER) en prenant soin de bloquer puis débloquer un *mutex*.

Théoriquement, un *thread* de recherche est bloqué lors de l'envoi d'une clause tant que cette dernière n'est pas reçue par les autres ordinateurs. Toutefois, en pratique, un tampon est utilisé afin de permettre aux solveurs d'envoyer des clauses sans attendre la fin de la communication. Ainsi, l'opération `MPI_SEND` est bloquée uniquement lorsque ce tampon est plein. Cela permet aux solveurs de se concentrer sur la recherche et pas sur l'envoi des clauses.

Concernant le *thread* receveur de chaque ordinateur, il reçoit les clauses les unes après les autres en utilisant l'opération bloquante `MPI_RECEIVE`. Par conséquent, ce *thread* effectue une attente active tant

que le message n'a pas été reçu. L'attente active peut alors diminuer la rapidité des solveurs séquentiels associés. En effet, dans cette situation, le *thread* receveur lit dans le fichier descripteur du *socket* réseau tant qu'un message n'a pas été reçu. Le nombre d'opérations alors effectué par le processeur peut alors ralentir les autres *threads*, notamment, ceux dédiés à la recherche. Heureusement, nous avons observé que cette situation n'arrive quasiment jamais. En effet, le nombre de clauses à partager sur le réseau est gigantesque car elles proviennent de tous les *threads* de recherche et que chacun d'entre eux peut produire un nombre de clauses apprises exponentiel par rapport à la taille de la formule initiale. Par conséquent, les *threads* receveurs ne font que recevoir des clauses et réalisent que très peu d'attente active. De plus, notons que malgré son défaut d'efficacité, une attente active permet de recevoir les clauses plus rapidement qu'une communication ne la réalisant pas.

Pour finir la description de ce modèle, notons aussi que nous devons limiter d'une manière heuristique le nombre de clauses à envoyer. Pour cela, l'heuristique utilisée est celle par défaut dans SYRUP (voir section 4.1.8).

7.5.2 Le problème des interblocages sur le réseau

Le lecteur intéressé par ce problème peut lire la section 3.6.2 de l'état de l'art. Cette dernière apporte plus de précision sur les problèmes de *deadlocks*, qui peuvent subvenir lors de l'utilisation de la mémoire partagée ou du passage de messages.

Comme il n'y a aucun cycle de communication, nous devons utiliser une autre manière d'éviter les *deadlocks* (interblocages). Pour cela, nous utilisons la notion de *thread safety*. Celle-ci assure que les *threads* exécutant des opérations `MPI_RECEIVE` et/ou `MPI_SEND` n'entraînent jamais de *deadlocks* quels que soient leurs ordres d'exécutions (Balaji *et al.* 2010). Cela est garanti par un algorithme distribué d'exclusion mutuelle (Singhal 1993, Lodha et Kshemkalyani 2000, Nishio *et al.* 1990, Wu *et al.* 2015) qui s'assure que seulement un *thread* peut exécuter une section critique à un temps donné. Le surcoût engendré par la notion de *thread safety* a été largement étudié dans Thakur et Gropp (2007). Néanmoins, la plupart des implémentations de la norme MPI ne possèdent pas d'algorithmes assez sophistiqués. En effet, à partir d'un certain nombre de cœurs de calcul, le *thread safety* n'est plus assuré par certaines implémentations. Par exemple, avec l'implémentation OpenMPI, nous notons l'apparition de *deadlocks* par passage de messages à partir de 32 cœurs de calcul. Pourtant, nous avons trouvé une implémentation nommée MPICH qui fonctionne bien et supporte le *thread safety* avec un grand nombre de cœurs de calcul. Des recherches sont encore d'actualité afin d'améliorer le *thread safety*. Par exemple, dans Grant *et al.* (2015), les auteurs détaillent un nouveau mécanisme conceptuel pour MPI afin de faire face à ce problème. À notre connaissance, aucun solveur SAT parallèle distribué n'est basé sur ce modèle de programmation.

7.5.3 Expérimentations

Toujours en utilisant le même protocole que pour les autres modèles de programmation (sections 7.3 et section 7.4), nous évaluons l'impact du modèle de programmation complètement hybride sur les performances globales du solveur. Le tableau 7.4 expose les résultats. Rappelons que comme les clauses sont envoyées dès que possible, il n'y a aucun temps de cycle de communication dans ce tableau pour le modèle complètement hybride. Par conséquent, nous pouvons juste examiner l'impact des différentes tailles du BUFFER. De plus, afin de comparer les différents modèles, nous avons choisi d'afficher à coté des résultats du modèle complètement hybride les deux meilleures configurations des modèles partiellement hybrides et purs par passage de messages. Pour ces deux derniers modèles, la meilleure configuration

Modèle de programmation	**Complètement hybride**			**Partiellement hybride**	**Pur passage de messages**
Taille du BUFFER	**20MB**	**100MB**	**200MB**	**20MB**	**20MB**
Reçues	35,776	32,840	33,343	38,448	30,989
Retenues	83%	86%	86%	68%	25%
Ret. (conflicts)	477	445	473	318	55
Ret. (useful)	5,845	5,899	6,184	3,256	618
Propagations	626,658	629,126	614,832	636,850	489,289
Conflits	1,344	1,246	1,268	1,364	823

TABLE 7.4 – Modèle de programmation complètement hybride. Impact de la taille des BUFFER sur les performances globales des solveurs. Tous ces résultats correspondent à la moyenne obtenue par tous les processus par seconde sur 20 instances variées de la SAT race 2015. Le temps des cycles de communication est de 5 secondes pour le modèle partiellement hybride et le modèle pur par passage de messages tandis qu'il n'y en a pas pour le modèle complètement hybride car ce dernier envoie les clauses dès que possible. Une focalisation est faite sur une taille du BUFFER fixée à 20MB.

était celle avec un temps des cycles de communication de 5 secondes et une taille du BUFFER fixée à 20MB (tableaux 7.2 et 7.3).

Ici encore, le meilleur compromis est d'avoir un BUFFER à 20 MB pour le modèle de programmation complètement hybride. De plus, nous observons des taux de propagations et de conflits par seconde comparables au modèle de programmation partiellement hybride. Plus intéressant, le nombre de clauses retenues utiles et en conflit (lignes 5 et 6) est beaucoup plus élevé pour le modèle complètement hybride. En d'autres termes, l'échange des clauses de ce modèle est plus efficace car celles-ci sont plus utiles à la recherche que pour les autres modèles. En effet, quand nous nous focalisons sur un BUFFER de 20 MB, le modèle complètement hybride à 477 clauses reçues en conflits par seconde tandis que celui partiellement hybride en possède que 318. Mieux encore, le taux des clauses reçues et utiles dans la recherche (utilisées dans une analyse de conflit afin d'apprendre une nouvelle clause) a doublé (5,845 pour le complètement hybride contre 3,256 pour le partiellement hybride).

Notre but est donc atteint : maintenir de bonnes performances globales et améliorer le partage des clauses afin que celles-ci soient plus utiles à la recherche. Dans la prochaine section, nous comparons expérimentalement l'efficacité des modèles de programmation afin de montrer que celui complètement hybride surpasse les deux autres.

7.6 Expérimentations

Dans la première partie de cette section, nous comparons les trois différents modèles de programmation sur les 100 instances de la SAT Race de 2015 (*parallel track*) afin d'ajuster notre solveur et de montrer quel est le meilleur modèle de programmation. Ensuite, nous comparons le meilleur modèle de programmation trouvé avec les solveurs SAT *portfolio* distribués TOPOSAT et HORDESAT sur les 300 instances fournies par la compétition SAT de 2016.

7.6.1 Comparaison des différents modèles de programmation

D'abord, nous comparons les trois modèles de programmation : le pur par passage de messages, le partiellement hybride et le complètement hybride sur les 100 instances de la compétition SAT-*race* 2015

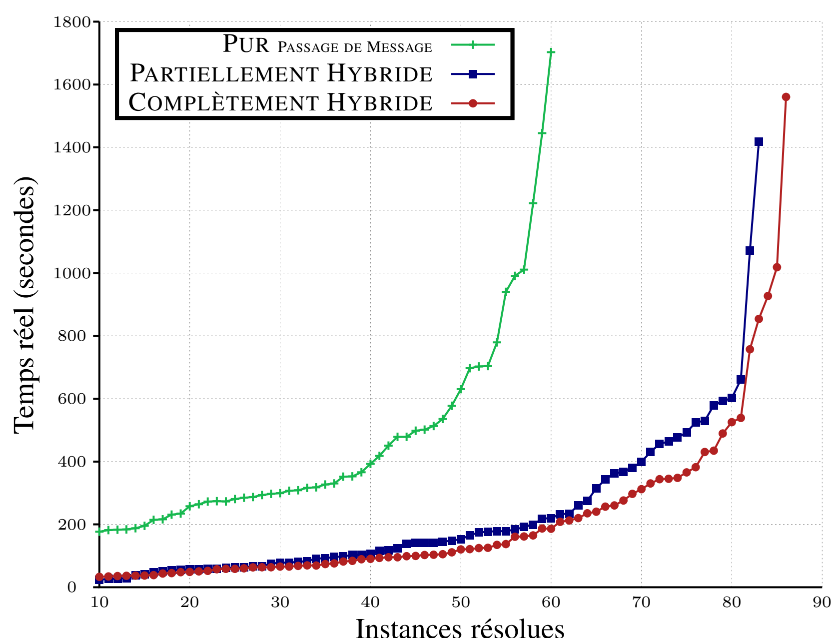


Figure 7.7 – Résultats expérimentaux des trois différents modèles de programmation (*cactus plot*) sur les 100 instances du *parallel track* de la SAT-*race* 2015. Les temps (axe des ordonnées) des instances résolues (axe des abscisses) sont triés du plus petit au plus grand.

Modèle de programmation	SAT	UNSAT	Total
Pur par passage de messages	25	26	61
Partiellement hybride	51	33	84
Complètement hybride	54	33	87

Table 7.5 – Résultats des trois différents modèles de programmation en nombre d'instances résolues.

du *track* parallèle. Comme pour les sections précédentes 7.5, 7.4 et 7.3, nous utilisons 32 ordinateurs contenant chacun 8 cœurs de calcul (32 bi-processors Intel XEON X7550 à 2 GHz avec un gigabyte Ethernet controller). Ces machines nous apportent donc un total de 256 cœurs de calcul. Le temps réel maximum accordé à la résolution est de 1,800 secondes pour chaque instance. Nous utilisons la meilleure configuration trouvée dans les sections précédentes (7.5, 7.4 et 7.3). Ces configurations ont alors des temps de cycle de communication de 5 secondes pour le modèle pur par passage de messages et le modèle partiellement hybride et une taille du BUFFER à 20MB pour tous les modèles. Les résultats sont reportés dans la figure 7.7 et le tableau 7.5.

Nous pouvons observer que ces résultats concordent avec ceux des sections précédentes et nos théories sur leurs performances. Notamment, le modèle pur par passage de messages est le moins bon des modèles en terme de performance. Il est seulement capable de résoudre 61 instances tandis que le modèle partiellement (resp. complètement) hybride résout 84 (resp. 87) instances. La principale raison est que le modèle pur par passage de messages réplique beaucoup trop de données. Notons que sur le même matériel, le solveur Syrup résout 57 instances avec seulement 8 cœurs de calcul. Par conséquence, nous montrons l'importance des modèles de programmation. En effet, une approche distribuée peut devenir très inefficace à cause des coûts de communication ! Cela explique pourquoi le solveur TopoSAT, basé

sur le modèle pur par passage de messages, essaye de réduire le nombre de clauses partagées via des topologies de communication afin de rester efficace.

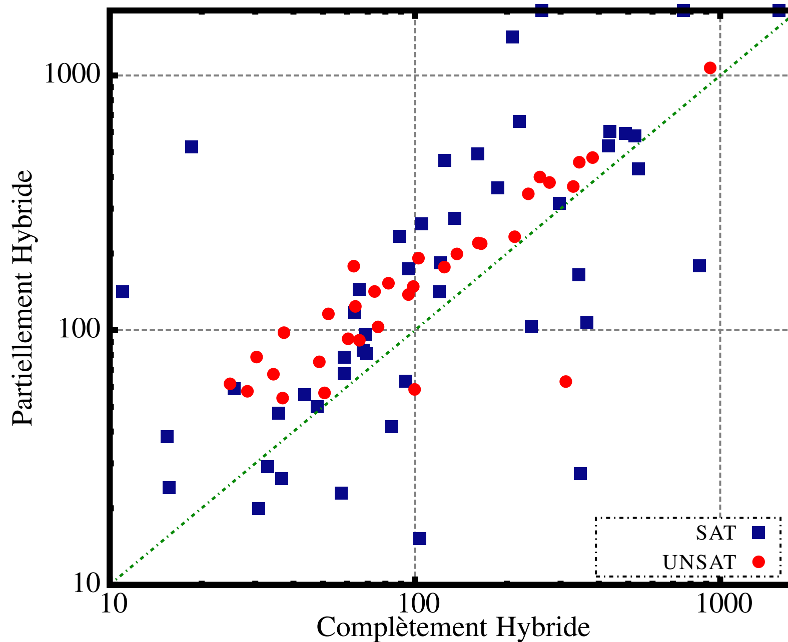


FIGURE 7.8 – Comparaisons expérimentales des modèles de programmation complètement et partiellement hybride (*scatter plot*) sur les 100 instances du *parallel track* de la SAT Race 2015. Un point représente les résultats d'une instance en temps de résolution (SAT et UNSAT).

À présent, nous effectuons une comparaison des modèles partiellement et complètement hybrides. En terme d'instances résolues (tableau 7.5), ces deux modèles résolvent le même nombre d'instances insatisfaisables (33 instances). Cela peut être expliqué par le fait qu'il ne reste probablement que très peu d'instances insatisfaisables à résoudre. En effet, lors de la compétition 2015, 33 instances ont été résolues insatisfaisables via les solveurs participants à cette compétition. Par conséquent, les deux modèles résolvent chacun la totalité des instances insatisfaisables déjà résolues à cette compétition. Un nouvel ensemble d'instances plus difficiles sera donc pris en compte dans nos travaux futurs. Remarquons aussi, que le modèle complètement hybride résout 3 instances satisfaisables de plus que celui partiellement hybride.

En terme de temps de résolution (figure 7.8 et 7.9), nous pouvons observer que le modèle complètement hybride est significativement le meilleur. Plus précisément, nous montrons que les instances sont résolues plus rapidement avec le modèle de programmation complètement hybride, excepté quelques instances, la plupart d'entre elles satisfaisables (familles « BITDIMACS », « JGIRALDEZ » et « 00X »).

Tous ces résultats valident nos études expérimentales préliminaires réalisées dans les sections précédentes. Cela valide également que plus nous avons de clauses retenues, plus les performances du solveur sont élevées. Par conséquent, il apparaît évident que pour augmenter le nombre de clauses partagées utiles, une bonne pratique est de les partager dès que possible. Nous appelons D-SYRUP notre solveur distribué et basé sur le modèle complètement hybride.

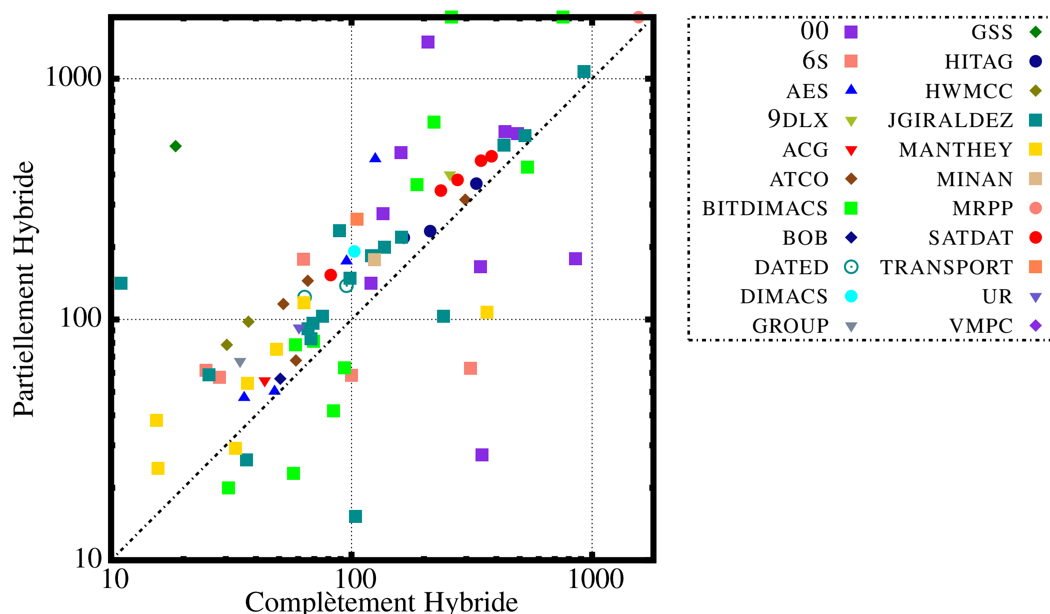


FIGURE 7.9 – Comparaisons expérimentales des modèles de programmation complètement et partiellement hybride (*scatter plot*) sur les 100 instances du *parallel track* de la SAT Race 2015. Un point représente les résultats d'une instance en temps de résolution (par famille).

7.6.2 Évaluation de D-SYRUP

Dans cette section, nous comparons la version complètement hybride de SYRUP, nommé D-SYRUP, contre le solveur partiellement hybride HORDESAT qui est composé de solveurs PLINGELING et contre le modèle pur par passage de messages TOPOSAT qui est composé de solveurs GLUCOSE (version 3). Tout ces solveurs utilisent 256 *threads*, c'est-à-dire, 32 ordinateurs possédant 8 cœurs chacun. Le matériel utilisé est le même que la section précédente (32 bi-processors Intel XEON X7550 à 2 GHz avec un gigabyte Ethernet controller). Afin d'être plus juste et montrer que notre solveur est efficace sur d'autres types d'instances que celles sur lesquelles nous l'avons développé, nous changeons les instances à résoudre. Au lieu de prendre celles de la section précédente (SAT race 2015), cette fois, nous utilisons les 300 instances provenant de la SAT compétition de 2016 (*application track*). Étant donné le nombre important d'instances, nous utilisons un temps limite à la résolution de 900 secondes. De plus, afin d'apporter une évaluation complète de D-SYRUP, nous avons aussi ajouté les résultats du solveur séquentiel GLUCOSE (1 cœur), du solveur parallèle *multi-thread* SYRUP (8 cœurs) et d'une version de notre solveur distribué D-SYRUP avec 128 cœurs (16 machines de 8 cœurs). Cela a pour but de voir si D-SYRUP maintient une bonne extensibilité sur cet ensemble d'instances.

Les résultats sont reportés sur la figure 7.10 et le tableau 7.6. Clairement, sur cet ensemble d'instances, D-SYRUP surpasse HORDERSAT et TOPOSAT. HORDERSAT semble pénalisé (figure 7.11) par des cycles de communication trop courts (1 seconde) entraînant trop d'accès dans les sections critiques. De plus, le BUFFER$_{network}$ d'HORDERSAT d'une taille fixe limite le partage (section 7.4). Concernant TOPOSAT, il est clairement inefficace (figure 7.12) car il est basé sur le modèle pur par passage de messages. Ces résultats montrent aussi que la stratégie paresseuse à la base de SYRUP maintient son efficacité quel que soit le nombre de cœurs utilisés. Rappelons que les clauses partagées par SYRUP sont celles partagées à toutes les machines avec D-SYRUP.

Quand nous comparons D-SYRUP avec 128 et 256 cœurs de calcul, dans la figure 7.10, le tableau 7.6

Solveur	**#Cœurs**	**SAT**	**UNSAT**	**Total**
D-SYRUP	256	75	109	184
HORDESAT	256	70	84	154
TOPOSAT	256	69	58	127
D-SYRUP	128	73	104	177
SYRUP	8	56	75	131
GLUCOSE	1	49	57	106

TABLE 7.6 – Résultats de GLUCOSE (1 cœur), SYRUP (8 cœurs), D-SYRUP (128 et 256 cœurs), HORDESAT (256 cœurs) et TOPOSAT (256 cœurs) sur les 300 instances de la compétition SAT 2016.

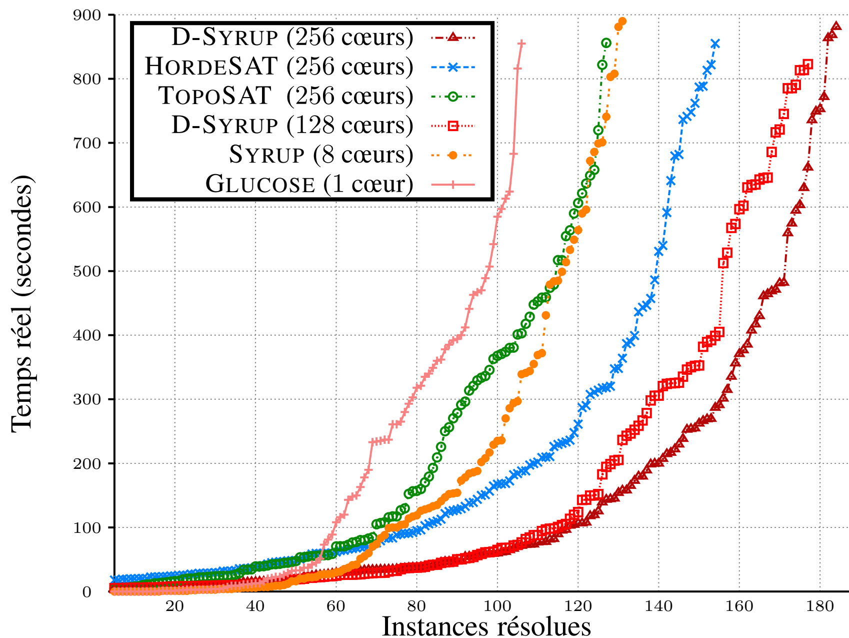


FIGURE 7.10 – Résultats de GLUCOSE (1 cœur), SYRUP (8 cœurs), D-SYRUP (128 et 256 cœurs), HORDESAT (256 cœurs) et TOPOSAT (256 cœurs) sur les 300 instances de la compétition SAT 2016 (*cactus plot*).

et le *scatter plot* 7.13, nous observons que notre solveur a une bonne extensibilité en nombre d'instances résolues. En effet, quand nous comparons SYRUP sur 8 cœurs à D-SYRUP, nous pouvons noter que D-SYRUP résout 53 instances de plus, en d'autres termes, c'est environ 18% d'instances résolues en plus. Notons que la version *multi-core* de SYRUP résout 154 instances (62 satisfaisables et 82 insatisfaisables) sur une machine de 32 cœurs de calcul (*quad-processor Intel XEON X7550*). Néanmoins, les accélérations des instances déjà résolues séquentiellement par GLUCOSE sont sous-linéaires, voir décroissantes pour certaines. Comme le montre la figure 7.13, une instance résolue en moins de 50 secondes avec 128 cœurs a peu de chance d'être résolue plus rapidement avec 256 cœurs. De plus, si nous prenons en compte le nombre de cœurs, la majorité des instances ont des accélérations sous-linéaires. Notons bien que cela n'est pas un problème propre à notre solveur. Avec un très grand nombre de cœurs, le nombre d'instances résolues augmente mais le *speedup* des instances déjà résolues avec un nombre de cœurs moins élevé est plus que sous-linéaire. En moyenne, l'extensibilité sur les instances résolues séquentiellement par GLUCOSE est représentée par des accélérations de 1.65, 3.60 et 4.15 pour, respectivement, un

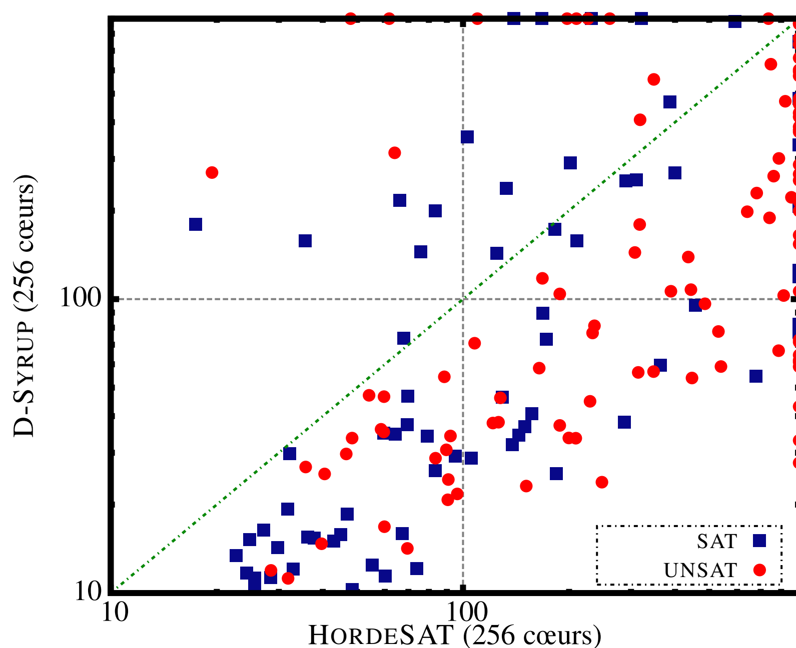


FIGURE 7.11 – HORDESAT *Versus* D-SYRUP (256 cœurs) (*scatter plot*). Chaque point représente une instance. Les points en dessous de la diagonale correspondent aux instances résolues plus rapidement par D-SYRUP (256 cœurs).

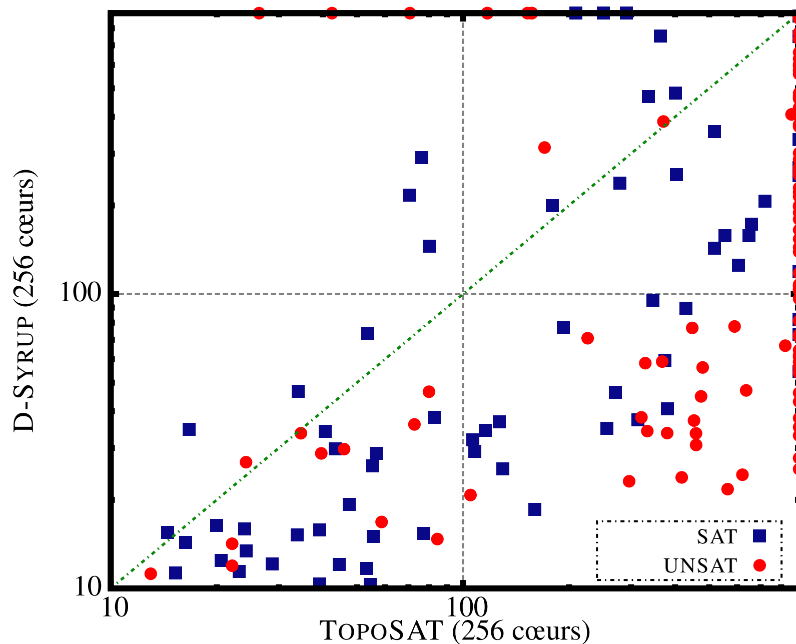


FIGURE 7.12 – TOPOSAT *Versus* D-SYRUP (256 cœurs) (*scatter plot*). Chaque point représente une instance. Les points en dessous de la diagonal correspondent aux instances résolues plus rapidement par D-SYRUP (256 cœurs).

nombre de cœurs de 8, 128 et 256.

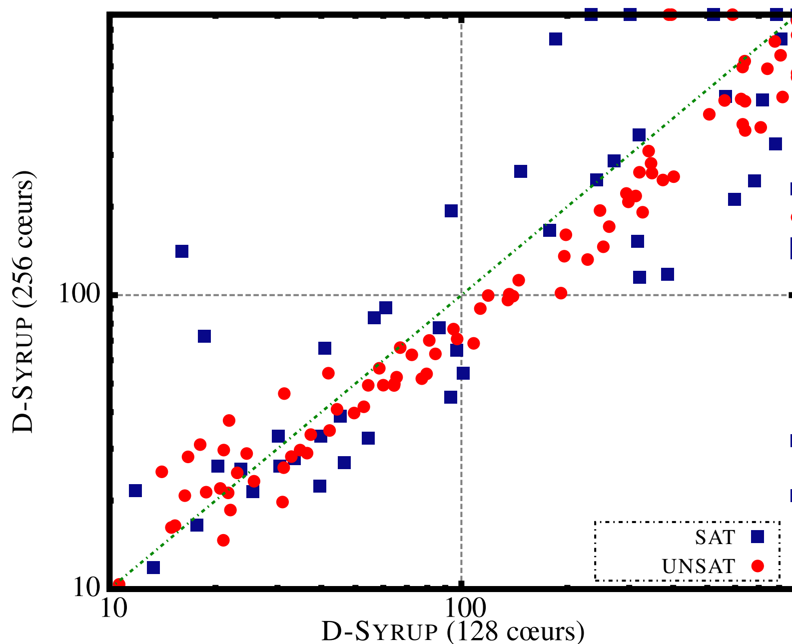


FIGURE 7.13 – D-SYRUP (128 cœurs) *Versus* D-SYRUP (256 cœurs) (*scatter plot*). Chaque point représente une instance. Les points en dessous de la diagonal correspondent aux instances résolues plus rapidement par D-SYRUP (256 cœurs).

7.7 Conclusion

Dans ces travaux, nous avons implémenté et évalué une version distribuée du solveur parallèle *multi-core* SYRUP. À cette fin, nous avons développé plusieurs modèles de programmation afin de gérer les communications. Plus précisément, nous avons proposé trois versions de SYRUP (pur par passage de messages, partiellement hybride et complètement hybride) en les configurant avec les meilleurs paramètres. Notamment, nous avons implémenté les modèles de programmation couramment utilisés dans les solveurs SAT distribués de l'état de l'art et utilisant des cycles de communication (pur par passage de messages et partiellement hybride). Cela nous a permis de montrer que plus la fréquence des cycles de communication est élevée, moins nous avons de propagation unitaire par seconde.

Grâce à cette observation, nous avons proposé un modèle complètement hybride qui est capable de partager les clauses sans cycle de communication. Nous avons démontré empiriquement que ce schéma permet de partager plus de clauses sans pénaliser le solveur. En effet, dans notre nouveau modèle complètement hybride, le nombre de clauses partagées et utiles à la recherche est doublé par rapport à celui partiellement hybride sans que le nombre de propagations par seconde ne soit affecté. De plus, remarquons que ce modèle ne nécessite pas d'ajuster le temps d'un cycle de communications, car il n'en possède pas. De surcroît, nous avons comparé ces versions afin de montrer que le modèle complètement hybride est le meilleur choix. En effet, ce modèle de programmation surpasse significativement les solveurs SAT de l'état de l'art TOPOSAT et HORDESAT sur un vaste ensemble d'instances.

Dans des expérimentations préliminaires, nous avons montré que la taille du tampon utilisé dans SYRUP afin d'échanger les clauses via la mémoire partagée a un impact important. En effet, quand le tampon est plein, les clauses ne sont plus partagée. Ainsi, nous avons testé trois tailles de tampon différentes (20MB, 100MB, et 200MB). Dans ces expérimentation, nous montrons qu'à partir d'une certaine taille de tampon, le nombre de clauses que le solveur doit gérer augmente tandis que le nombre

de propagations réalisées par seconde diminue. Plus précisément, une taille du tampon de 20MB est la meilleure option. En effet, elle permet de ne pas surcharger les solveurs.

Nos expérimentations semblent montrer que partager les clauses aussi vite que possible entraîne un impact positif sur l'efficacité des solveurs. Cependant, nous avons aussi montrer que SYRUP et D-SYRUP ont une extensibilité en moyenne super-linéaire avec 8 cœurs de calcul mais qui décroît vers une extensibilité sous-linéaire avec plus de cœurs. Toutefois, le nombre d'instances résolues augmente toujours (la version D-SYRUP de 256 cœurs résout 7 instances de plus que celle de 128 cœurs).

Conclusion et perspectives

Dans le cadre de cette thèse, nous avons tout d'abord étudié deux approches afin de diviser une instance du problème SAT en sous-problèmes afin de développer une approche « diviser pour mieux régner » : l'une (UCTSAT) est basée sur le dilemme exploration/exploitation en utilisant l'algorithme UCT et l'autre (SWARMSAT) transforme la décomposition statique du solveur CUBEANDCONQUER en une décomposition dynamique (chapitre 5). Expérimentalement, ces deux approches n'ont pas obtenues les résultats attendus. Cependant, elles nous ont permis de mettre en exergue qu'une décomposition dynamique du problème en sous-problèmes était plus efficace sur les instances considérées.

Suite à ces observations, nous avons développé une nouvelle décomposition dynamique du problème initial en sous-problèmes dans un solveur nommé AMPHAROS (chapitre 6). L'originalité de ce solveur est que la décomposition dynamique s'adapte suivant plusieurs paramètres :

- le nombre de clauses partagées et redondantes ;
- le nombre de sous-problèmes déjà généré.

En effet, quand les clauses échangées sont redondantes, nous augmentons la décomposition afin de diversifier la recherche et ainsi produire moins de clauses redondantes. A contrario, si les clauses sont convenablement diversifiées, nous intensifions la recherche sur les sous-problèmes déjà présents. De plus, nous limitons d'une manière heuristique le nombre de sous-problèmes afin de pas explorer trop en profondeur l'arbre de recherche. Autre point important, dans AMPHAROS, plusieurs solveurs peuvent travailler en même temps sur le même sous-problème et les solveurs ne restent jamais très longtemps sur la résolution d'un sous-problème. Plus précisément, ils ont la possibilité de changer régulièrement de sous-problèmes afin de ne pas rester éternellement sur un sous-problème trop « difficile ».

Cette manière de réaliser la décomposition dynamique a eu pour effet d'améliorer l'efficacité de notre approche « diviser pour mieux régner », mais aussi d'augmenter la fréquence d'une nouveau type de partage : les littéraux unitaires sous hypothèses. Ces littéraux sont dépendant d'un sous-problème et sont partagés lorsqu'un solveur change de sous-problème. Pendant cet échange, certaines techniques *inprocessing* permettent de déduire des nouvelles informations qui sont encore des littéraux unitaires sous hypothèses. Les expérimentations sur AMPHAROS, quelles soient réalisées sur ses composants, sur son extensibilité ou sur sa comparaison avec des solveurs de l'état de l'art, démontrent qu'AMPHAROS est un solveur « diviser pour mieux régner » très performant.

Cependant, même si AMPHAROS a obtenu de très bons résultats lors des expérimentations, nous mettons en évidence que l'encombrement du réseau par l'échange des clauses a un impact important sur son efficacité. Pour remédier à ce problème, nous avons étudié l'impact des modèles de programmation parallèle sur cet échange. Ainsi, Les travaux induis ont apporté une solution novatrice aux possibles congestions du réseau dans un environnement massivement distribué.

À cette fin, trois modèles de programmation ont été mis en œuvre dans un solveur nommé D-SYRUP (chapitre 7). Deux d'entre eux ont déjà été implémentés dans l'état de l'art : le modèle de programmation pur par passage de messages (TOPOSAT, section 4.1.10, Ehlers *et al.* (2014)) et le partiellement hybride (HORDESAT, section 4.1.11, Balyo *et al.* (2015)). Nous avons alors montré que le modèle pur est le moins bon et nous avons amélioré sensiblement celui partiellement hybride. De surcroît, nous avons remarqué sur ces deux modèles de programmation que le temps dédié à un cycle de communications a un

impact très important sur l'efficacité du solveur. À partir de ces informations, nous avons implémenté un modèle de programmation distribué dit complètement hybride permettant d'envoyer les clauses plus rapidement, plus précisément, dès que possible. Notons que ce modèle n'a, à notre connaissance, jamais été développé dans le cadre de SAT. Via des expérimentations, nous avons comparé ces versions afin de montrer que le modèle complètement hybride est le meilleur choix. En effet, nous avons expérimentallement démontré que ce modèle de programmation surpasse significativement les solveurs SAT parallèle de l'état de l'art TOPOSAT et HORDESAT sur un vaste ensemble d'instances.

Plus généralement, en observant la totalité des contributions de ce manuscrit, nous pouvons observer que l'extensibilité d'un solveur SAT parallèle décroît à partir d'un certains nombre de cœurs. Plus précisément, pour le solveur SYRUP, au delà de 8 cœurs de calcul, les accélérations passent de super-linéaire à sous-linéaire. De plus, dans D-SYRUP, l'extensibilité sur 256 cœurs montre qu'il n'est pas utile d'employer autant de cœurs sur les instances déjà résolues séquentiellement. Cela peut être expliqué par les lois d'Amdahl de Gustafson. La première dit que la partie de code séquentiel limite les accélérations. En effet, dans le cadre de SAT, il est fort probable que la complexité et la performance des solveurs séquentiels restreint la partie de code parallèle réalisée en fonction de l'approche considérée. De plus, nous pouvons observer que la loi de Gustafson semble être appliquée. En effet, les problèmes qui contiennent plus de données en entrées (clauses et variables) ont plus de « chance » d'être résolues grâce au parallélisme. Pourtant, malgré quelques accélérations sous-linéaires sur certains problèmes, d'autres, notamment, non résolues séquentiellement, le sont en parallèle, grâce à l'utilisation d'un grand nombre d'unités de calcul (256 cœurs). À présent, nous allons décrire quelques perspectives.

La première est, sans nul doute, la plus urgente par rapport à nos contributions. Il s'agit de reprogrammer AMPHAROS en décentralisant ses communications et en utilisant le modèle de programmation complètement hybride. Cela devrait significativement augmenter les performances d'AMPHAROS. Toutefois, le temps nécessaire à la programmation du modèle complètement hybride est plus long quand il est associé à l'approche « diviser pour mieux régner ». Même si la plus grand partie des informations à échanger sont les clauses apprises, des risques supplémentaires de *deadlock* sont induis par les techniques utilisées dans l'approche « diviser pour mieux régner ». En effet, dans cette approche, ces différentes techniques (gestion des sous-problèmes, décomposition dynamique, littéraux unitaires sous hypothèses, ...) requièrent quelques communications en plus.

Une autre perspective est axée sur le fait que les solveurs SAT distribués sont très compliqués à mettre en place sur une architecture particulière. En effet, les *clusters* n'ont pas toujours la bonne configuration logicielle. Plus précisément, même si une norme MPI existe, elle peut encore changer en fonction des versions installées sur le *cluster* considéré. De plus, les implémentations MPI ont des performances différentes. À titre d'exemple, nous avons directement observé ce problème avec AMPHAROS à la compétition SAT 2016. En effet, il a été nécessaire de reprogrammer une partie du code car certaines routines de programmation n'était pas prises en charge. Les routines manquantes ont eu pour effet de diminuer sensiblement les performances d'AMPHAROS car certaines fonctionnalités n'étaient pas prises en charge par le système. Une solution à ce problème est d'apporter des conteneurs *dockers* afin de simplifier le déploiement d'un solveur et ainsi faciliter son installation et son utilisation. Ces conteneurs contiennent les bonnes versions des logiciels nécessaires au bon fonctionnement d'un programme. Par conséquent, un programme associé à cette technologie peut être exécuté sur n'importe quelle architecture matérielle beaucoup plus facilement.

Une autre direction de recherche est d'explorer des concepts plus avancés en relation avec la gestion des clauses afin de faire une meilleure sélection des clauses à supprimer dans le tampon de SYRUP (section 4.1.8) avant qu'il devienne plein. Une première tentative pourrait être de trier les clauses suivant leurs tailles ou une autre mesure (PSM par exemple) et ainsi garder les clauses les plus courtes en priorité. Alternativement, nous pouvons construire une stratégie qui sélectionne avec soin les clauses à

supprimer. Par exemple, un critère de sélection pourrait être de garder seulement les clauses binaires quand la capacité du tampon est presque pleine ou supprimer les clauses d'une taille supérieure à 8 si le tampon est à moitié plein.

Afin de continuer dans la direction des travaux réalisés pour D-SYRUP, nous allons essayer de retarder la transmission des clauses. Si ce retard engendre bien une baisse de performance, nous aurons la possibilité d'utiliser une heuristique basée sur le temps passé depuis les créations des clauses afin de supprimer les clauses du tampon. Pour cela, nous pourrions inclure le moment de la création d'une clause apprise dans les données à échanger lors du partages des clauses. Ainsi, quand un solveur reçoit des clauses, il pourra les trier en fonction de leur âge et les plus veilles pourront être écartées si celui-ci est plein.

Une autre idée est basée sur le fait qu'il peut exister des heuristiques calculées différemment afin d'évaluer le même objet, comme par exemple dans les solveurs parallèles SYRUP et D-SYRUP. D'un côté, l'heuristique LBD est utilisée afin de supprimer des clauses apprises. Pour rappel, cela permet d'éviter un ralentissement du solveur pendant la recherche dû à la propagation unitaire des littéraux des clauses apprises. Ce qui est important dans cette perspective est que cette heuristique permet donc d'estimer la qualité des clauses. D'un autre coté, une autre heuristique est utilisée dans le partage des clauses et elle aussi, permet d'estimer la qualité des clauses afin de partager uniquement celles qui semblent utiles à la recherche (voir section 4.1.8). Par conséquent, nous avons deux heuristiques ayant le même but mais celles-ci ne sont pas utilisées pour réaliser la même chose. Dans le cadre de SAT, il est bien connu que certaines heuristiques fonctionnent mieux que d'autres sur quelques problèmes. L'idée de cette perspective est de garder la même heuristique afin de supprimer les clauses apprises et de les partager. Ainsi, le but est de ne pas supprimer les clauses que l'on partage ou vise-versa. De plus, nous pensons pouvoir utiliser plusieurs heuristiques grâce à un Monté Carlo. Plus précisément, voir le choix d'une heuristique jugeant la qualité des clauses pour une instance donnée comme un problème de Monté Carlo. Pour réaliser cela, chaque heuristique devra renvoyer un gain qui représentera la qualité de son utilisation durant une certaine période. Par la suite, une moyenne glissante exponentielle (section définition 2.22) ou le calcul des valeurs UCB (chapitre 5) décidera la prochaine heuristique à choisir pour un certain laps de temps. Par conséquent, nous traiterions le dilemme exploration/exploitation sur ces heuristiques. En d'autres termes, il est représenté par cette question : devons nous exploiter une heuristique qui semble performante dans sa tache ou devons nous explorer une autre heuristique afin de l'évaluer ? Remarquons qu'une idée similaire (Paparrizou et Stergiou 2012) a déjà été réalisée en programmation par contraintes (CP pour *constraint programming*) afin de choisir dynamiquement un propagateur de contrainte.

Récemment, une technique *inprocessing* a obtenu de très bons résultats séquentiellement (Luo *et al.* 2017). Toutefois, une perspective peut être d'appliquer certaines techniques *inprocessing* dans un environnement parallèle. En d'autres termes, ces techniques pourraient être réalisées par des unités de calcul dédiées. Cela a déjà été réalisé dans les travaux de Wieringa et Heljanko (2013) (section 4.1.7). Cependant, ces travaux sont récents et les auteurs proposent un solveur composé que de deux *threads* : un pour la recherche et un pour de l'*inprocessing*. Cette perspective a pour but de faire de l'*inprocessing* dans un environnement massivement distribué. Par exemple, nous pourrions dédier un *thread* par machine à l'*inprocessing*. Cela permettrait de ne pas ralentir les *threads* de recherche et ainsi de pouvoir employer des méthodes plus coûteuses en temps de calcul. Remarquons qu'AMPHAROS fait déjà certaines techniques *inprocessing* via les littéraux unitaires sous hypothèses. L'idée est alors de trouver et de tester d'autres prétraitements qui peuvent être appliqués en parallèle et pendant la recherche.

Index

A

Accélération
- absolue . . . 72
- en latence . . . 77
- linéaire . . . 72
- relative . . . 72
- sous-linéaire . . . 72
- super-linéaire . . . 72

Affecter . . . 24

Algorithme
- parallèles . . . 71
- séquentiel . . . 71

Analyse de conflit . . . 34

Architecture
- ccNUMA . . . 69
- ccUMA . . . 69
- de Von Neumann . . . 60
  - entrée-sortie . . . 60
  - unité arithmétique et logique . . . 60
  - unité de contrôle . . . 60
- NUMA . . . 68
- UMA . . . 68

Assumption . . . 106

B

Backbone . . . 39
Backjump . . . 35
Backtrack . . . 31
Barrière . . . 92

C

Cœur de calcul . . . 66

Calcul
- parallèle . . . 57
- séquentiel . . . 57

Clause . . . 13
- n-aire . . . 13
- assertive . . . 34
- binaire . . . 13
- bloquée . . . 52
- de Horn . . . 16
- falsifiée . . . 13
- mixte . . . 13
- négative . . . 13
- positive . . . 13
- raison . . . 32
- reverse-Horn . . . 16
- satisfaite . . . 13
- subsumée . . . 13
- ternaire . . . 13
- unisatisfaite . . . 13
- unitaire . . . 13

CNF . . . 13
Cohérence du cache . . . 69
Communautés . . . 97
Composant d'une formule CNF . . . 41
Conditionner . . . 24
Conflict . . . 31
Conséquence logique . . . 12
Contre-modèle . . . 12
Cube . . . 13

D

Décision . . . 30
Décomposition dynamique . . . 100
Décomposition statique . . . 100
Déviation de la *Phase Saving* . . . 47
deadlock . . . 84
Degré de redondance . . . 142
Diversification . . . 88
DNF . . . 13

E

Echange d'information « centralisé » . . . 83
Efficacité . . . 72
Elagage dans AMPHAROS . . . 138
Équilibrage de charge . . . 74
Équivalence logique . . . 12

Exécution
- parallèle . . . 71
- séquentielle . . . 71

Exclusion mutuelle . . . 82
Explication(s) d'un littéral . . . 32
Extensibilité . . . 73
Extension dans AMPHAROS . . . 137

F

Flux

d'instructions 64
de données 64

Formule
insatisfaisable 12
propositionnelle 10
satisfaisable 12
valide 12

G

Granularité 80
Graphe d'implications 33

H

Heuristique de branchement
DLCS 39
Heuristique de branchement
DLIS 39
Heuristique de branchement
BOHM 39
EVSIDS 40
JW 40
MOMS 39
NVSIDS 40
VSIDS 40
Hyper-Binary-Resolution 53
Hyper-Unary-Resolution 53

I

Indéterminisme 91
Initialisation dans AMPHAROS 134
Instruction 57
Intensification 88
Interblocage 84
Interprétation 10
complète 12
partielle 12
prolongement 12

L

Latence 77
LBD 46
Littéral 11
Assumption 106
bloquée 52
complémentaire 11
négatif 11
positif 11
pure 28
Littéraux équivalents 91
Littéraux unitaires sous hypothèses 139
Logique propositionnelle
connecteurs usuels 11
formule 10
sémantique 11
vocabulaire 10
Loi
d'Amdahl 78
de Gustafson 78

M

Mémoire 60
cache 61
partagée 67
Modèle 12
Modèle de programmation
complètement hybride 166
partiellement hybride 163
pur par passage de messages 160
Modèles de programmation parallèle
MPMD 81
SPMD 81
Moyenne glissante exponentielle 42
Multi-Armed Bandit problem 93
Multiprocesseur symétrique 68

N

Nœud
dominant 34
précédent 33
suivant 33
Niveau de décision 30
NNF 13

P

Parallélisme 57
au niveau des instructions 62
de données 80
de tâches 80
Partage dans AMPHAROS 138
Pipeline 62
Point d'implication unique
F-UIP 34
L-UIP 34
Problème
NP-complet 14
k-SAT 16
2-SAT 16
3-SAT 16

des bandits manchots à k bras 93
parfaitement parallèle 75
de la coloration de graphes 18
des triplets pythagoriciens booléens 20
du carré latin 19
de décision 14
Horn-SAT 17
renameable-Horn-SAT 17
reverse-Horn-SAT 17
SAT 15
vérification de modèles bornés 17
Processeur
many-cœurs 70
multi-cœurs 66
Propagation unitaire 27
PSM 47

R

Résolution 23
unitaire 27
appliquée à une variable 25
de Davis et Putnam 25
Règle
de division 24
Registres 61

S

Séquence
de décisions 30
de décisions/propagations 30
de propagations 30
SAT 15
Sections critiques 82
Simplifier 24
Solveurs SAT parallèles
AMPHAROS 133
D-SYRUP 153
SWARMSAT 126
CBPENELOPE 97
CUBEANDCONQUER 106
DP^{2}LL 91
DOLIUS 110
GLUCORED 94
HORDESAT 99
MANYSAT 88
MAPLEAMPHAROS 150
MINIRED 94
PARACIRMINISAT 97
PART-TREE-LEARNING 106
PENELOPE 94
PSATO 101
SATCIETY 105
SYRUP 96
TOPOSAT 98
UCTSAT 115
BESS 93
C-SAT 104
MTSS 104
PLINGELING 90
PPFOLIO 92
Subsumption 13

T

Taxonomie de Flynn 64
MIMD 64
MISD 64
SIMD 64
SISD 64
Transmission dans AMPHAROS 135

U

UCB *Upper Confidence Bounds* 115
UCT *Upper Confidence Bounds Tree* 115

V

Virtual Best Solver 93

Bibliographie

Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95), volume 976 de *Lecture Notes in Computer Science*, Cassis, France, septembre 1995. Springer.

Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03), volume 2929 de *Lecture Notes in Computer Science*, Santa Margherita Ligure, Italy, mai 2003. Published in 2004. Springer.

Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05), volume 3569 de *Lecture Notes in Computer Science*, St. Andrews, Scottland, juin 2005a. Springer.

Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04), volume 3542 de *Lecture Notes in Computer Science*, Vancouver (BC) Canada, mai 10-13 2004. Revised selected papers published in 2005b. Sringer.

Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), Hyderabad, India, janvier 6-16 2007.

Martin AIGNER, Armin BIERE, Christoph M. KIRSCH, Aina NIEMETZ et Mathias PREINER : Analysis of portfolio-style parallel SAT solving on current multi-core architectures. *In POS@SAT*, volume 29 de *EPiC Series in Computing*, pages 28–40, 2013.

Gene M. AMDAHL : Validity of the single processor approach to achieving large scale computing capabilities. *In Proceedings of the Spring Joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM.

David P. ANDERSON, Jeff COBB, Eric KORPELA, Matt LEBOFSKY et Dan WERTHIMER : Seti@home : An experiment in public-resource computing. *Journal of Communications of the ACM*, 45(11):56–61, novembre 2002.

G. AUDEMARD, A. BIERE, J-M. LAGNIEZ et L. SIMON : Améliorer SAT dans le cadre incrémental. *RIA*, pages 593–614, 2014a.

G. AUDEMARD, B. HOESSEN, S. JABBOUR et C. PIETTE : An effective distributed d&c approach for the satisfiability problem. *In PDP*, pages 183–187, 2014b.

G. AUDEMARD, JM. LAGNIEZ et L. SIMON : Improving glucose for incremental SAT solving with assumptions : Application to MUS extraction. *In SAT*, pages 309–317, 2013a.

G. AUDEMARD, JM. LAGNIEZ et L. SIMON : Just-in-time compilation of knowledge bases. *In IJCAI*. IJCAI/AAAI, 2013b.

Gilles AUDEMARD, Lucas BORDEAUX, Youssef HAMADI, Saïd JABBOUR et Lakhdar SAÏS : A generalized framework for conflict analysis. *In Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 21–27, Berlin, Heidelberg, 2008a. Springer-Verlag.

Gilles AUDEMARD, Lucas BORDEAUX, Youssef HAMADI, Saïd JABBOUR et Lakhdar SAIS : A generalized framework for conflict analysis. *In SAT*, volume 4996 de *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008b.

Gilles AUDEMARD, Benoît HOESSEN, Saïd JABBOUR, Jean-Marie LAGNIEZ et Cédric PIETTE : Revisiting clause exchange in parallel SAT solving. *In Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 de *Lecture Notes in Computer Science*, pages 200–213. Springer, 2012.

Gilles AUDEMARD, Benoît HOESSEN, Saïd JABBOUR et Cédric PIETTE : Dolius : A distributed parallel SAT solving framework. volume 27 de *EPiC Series in Computing*, pages 1–11. EasyChair, 2014c.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE et Lakhdar SAÏS : Integrating conflict driven clause learning to local search. *In Proceedings of International Workshop on Local Search Techniques in Constraint Satisfaction (affiliated to CP) (LSCS'09)*, Lisbon, Portugal, septembre 2009. Electronic proceedings.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE et Lakhdar SAIS : Boosting local search thanks to cdcl. *In LPAR (Yogyakarta)*, volume 6397 de *Lecture Notes in Computer Science*, pages 474–488. Springer, 2010.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE et Lakhdar SAÏS : On freezeing and reactivating learnt clauses. *In Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, jun 2011.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Nicolas SZCZEPANSKI et Sébastien TABARY : Swarmsat : un solveur sat massivement parallèle. *In Journées Francophones de Programmation par Contraintes, JFPC'15*, 2015.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Nicolas SZCZEPANSKI et Sébastien TABARY : An adaptive parallel SAT solver. *In Principles and Practice of Constraint Programming - 22nd International Conference, CP'16, Proceedings*, pages 30–48, 2016a.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Nicolas SZCZEPANSKI et Sébastien TABARY : Ampharos : Un solveur sat parallèle adaptatif. *In Journées Francophones de Programmation par Contraintes, JFPC'16*, 2016b.

Gilles AUDEMARD, Jean-Marie LAGNIEZ, Nicolas SZCZEPANSKI et Sébastien TABARY : A distributed version of syrup. *In Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*, Lecture Notes in Computer Science, pages 215–232, 2017.

Gilles AUDEMARD et Laurent SIMON : Glucose : a solver that predicts learnt clauses quality. Rapport technique, 2009a. SAT 2009 Competition Event Booklet.

Gilles AUDEMARD et Laurent SIMON : Predicting learnt clauses quality in modern sat solver. *In Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009b.

Gilles AUDEMARD et Laurent SIMON : Refining restarts strategies for sat and unsat. *In Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 118–126, Berlin, Heidelberg, 2012. Springer-Verlag.

Gilles AUDEMARD et Laurent SIMON : Lazy clause exchange policy for parallel SAT solvers. *In SAT*, volume 8561 de *Lecture Notes in Computer Science*, pages 197–205. Springer, 2014.

Gilles AUDEMARD et Laurent SIMON : Extreme cases in SAT problems. *In Proceedings of the Nineteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 de *Lecture Notes in Computer Science*, pages 87–103. Springer, 2016.

Peter AUER et Ronald ORTNER : Ucb revisited : Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, 61(1-2):55–65, 2010.

Pavan BALAJI, Darius BUNTINAS, David GOODELL, William GROPP et Rajeev THAKUR : Fine-grained multithreading support for hybrid threaded MPI programming. *International Journal of High Performance Computing Applications IJHPCA*, 24(1):49–57, 2010.

A. BALINT, M. HEULE, A. BELOV et M. JARVISALO : The application and the hard combinatorial benchmarks in sat competition 2013. *In SAT'13*, page 99, 2013.

Tomas BALYO, Peter SANDERS et Carsten SINZ : Hordesat : A massively parallel portfolio SAT solver. *In Proceedings of the Eighteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 de *Lecture Notes in Computer Science*, pages 156–172. Springer, 2015.

Amotz BAR-NOY, Mihir BELLARE, Magnús M. HALLDÓRSSON, Hadas SHACHNAI et Tami TAMIR : On chromatic sums and distributed resource allocation. *Inf. Comput.*, 140(2):183–202, 1998.

Peter BARTH : A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization. Rapport technique, 1995.

Roberto J. BAYARDO JR. et Robert C. SCHRAG : Using csp look-back techniques to solve real-world sat instances. *In Proceedings of the Fourteenth American National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence (Rhode Island, USA), juillet 1997.

Frank E. BENNETT et Hantao ZHANG : Latin squares with self-orthogonal conjugates. *Journal of Discrete Mathematic*, 284(1-3):45–55, juillet 2004a.

Frank E. BENNETT et Hantao ZHANG : Latin squares with self-orthogonal conjugates. *Journal of Discrete Mathematics*, 284(1-3):45–55, juillet 2004b. ISSN 0012-365X.

D. Le BERRE : Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001a.

Daniel Le BERRE : Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9, 2001b.

A. BIERE : Yet another local search solver and lingeling and friends entering the sat competition 2014. *In Proc. of SAT competition*, 2014.

A. BIERE et A. FROHLICH : Evaluating cdcl restart schemes (preliminary version). *In In Workshop on Pragmatics of SAT (POS)*, 2015.

Armin BIERE : Adaptive restart strategies for conflict driven SAT solvers. *In SAT*, volume 4996 de *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008a.

Armin BIERE : Adaptive restart strategies for conflict driven sat solvers. *In* Hans KLEINE BÃ$\frac{1}{4}$NING et Xishun ZHAO, éditeurs : *Theory and Applications of Satisfiability Testing (SAT)*, volume 4996 de *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin / Heidelberg, 2008b.

Armin BIERE : Precosat system description. SAT Competition, solver description, 2009.

Armin BIERE : Lingeling, plingeling, picosat and precosat at sat race 2010, August 2010.

Armin BIERE : Lingeling and friends at the sat competition 2011. 2011.

Armin BIERE : Lingeling and friends entering the sat challenge 2012, 2012.

Armin BIERE : Lingeling, plingeling and treengeling entering the sat competition 2013, 2013.

Armin BIERE, Alessandro CIMATTI, Edmund M. CLARKE, Masahiro FUJITA et Yunshan ZHU : Symbolic model checking using sat procedures instead of bdds. *In Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

Armin BIERE et Andreas FRÖHLICH : Evaluating CDCL variable scoring schemes. *In Proceedings of the Eighteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 de *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015.

Armin BIERE, Marijn J. H. HEULE, Hans van MAAREN et Toby WALSH, éditeurs. *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, février 2009. ISBN 978-1-58603-929-5.

Armin BIERE et Carsten SINZ : Decomposing SAT problems into connected components. *JSAT*, 2 (1-4):201–208, 2006.

Vincent D. BLONDEL, Jean-Loup GUILLAUME, Renaud LAMBIOTTE et Etienne LEFEBVRE : Fast unfolding of community hierarchies in large networks. *CoRR*, 2008.

E. BOROS, Y. CRAMA et P. L. HAMMER : Polynomial-time inference of all valid implications for horn and related formulae. *Journal of Annals of Mathematics and Artificial Intelligence*, 1(1):21–32, Sep 1990.

Alfredo BRAUNSTEIN, Marc MÉZARD et Riccardo ZECCHINA : Survey propagation : An algorithm for satisfiability. *Random Struct. Algorithms*, 27:201–226, September 2005.

Micheal BURO et Hans KLEINE-BÜNING : Report on the sat competition. Rapport technique, University of Paderborn, 1992.

Thomas CARIDROIT, Jean-Marie LAGNIEZ, Daniel Le BERRE, Tiago de LIMA et Valentin MONTMIRAIL : A sat-based approach for solving the modal logic s5-satisfiability problem. *In AAAI*, pages 3864–3870. AAAI Press, 2017.

Patrick CARRIBAULT, Marc PÉRACHE et Hervé JOURDREN : Enabling low-overhead hybrid mpi/openmp parallelism with MPC. *In International Workshop on OpenMP IWOMP'10, Proceedings*, pages 1–14, 2010.

V. CHANDRU et J. N. HOOKER : Extended horn sets in propositional logic. *Journal of ACM*, 38(1):205–221, janvier 1991.

W. CHRABAKH et R. WOLSKI : The gridsat portal : a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation : Practice and Experience*, 19(6):795–808, 2007.

Thiffault CHRISTIAN, Bacchus FAHIEM et Walsh TOBY : Solving non-clausal formulas with DPLL search. *In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.

Geoffrey CHU, Peter J. STUCKEY et Aaron HARWOOD : Pminisat : A parallelization of minisat 2.0. Rapport technique, 2008.

CMAP, TAO et LRI : Programme mogo : http ://senseis.xmp.net/ ?mogo, 2006.

Stephen A. COOK : The complexity of theorem-proving procedures. *In STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

James M. CRAWFORD et Larry D. AUTON : Experimental results on the crossover point in satisfiability problems. *In Proceedings of the eleventh national conference on Artificial intelligence*, AAAI'93, pages 21–27. AAAI Press, 1993.

Mukesh DALAL : Efficient propositional constraint propagation. *In Proceedings of the Tenth American National Conference on Artificial Intelligence (AAAI'92)*, pages 409–414, San-Jose, California (USA), 1992.

Martin DAVIS, George LOGEMANN et Donald LOVELAND : A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5:394–397, 1962.

Martin DAVIS et Hilary PUTNAM : A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

Rina DECHTER : Bucket elimination : a unifying framework for processing hard and soft constraints. *Journal of Constraints*, 2(1):51–55, 1997.

Gilles DEQUEN et Olivier DUBOIS : kcnfs : An efficient solver for random k-sat formulae. *In Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)* pro (2004), pages 486–501.

Edsger W. DIJKSTRA : Solution of a problem in concurrent programming control. *Commun. ACM*, 8 (9):569, 1965.

M. DORIGO et T. STÜTZLE : *Ant colony optimization*. the MIT Press, 2004.

Vijay D'SILVA, Daniel KROENING et Georg WEISSENBACHER : A survey of automated techniques for formal software verification. *Journal of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

Olivier DUBOIS, Pascal ANDRÉ, Yacine BOUFKHAD et Jacques CARLIER : Sat versus unsat. *In* Johnson et Trick (1996), pages 415–436.

Niklas EÉN et Armin BIERE : Effective preprocessing in sat through variable and clause elimination. *In Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)* pro (2005a), pages 61–75.

Niklas EÉN et Niklas SÖRENSON : An extensible sat-solver. *In Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)* pro (2004), pages 333–336.

Niklas EÉN et Niklas SÖRENSON : Minisat 2.1 and minisat++ 1.0 sat race 2008 editions. Rapport technique, 2008.

Thorsten EHLERS, Dirk NOWOTKA et Philipp SIEWECK : Communication in massively-parallel SAT solving. *In ICTAI*, pages 709–716. IEEE Computer Society, 2014.

Victor EIJKHOUT : *Introduction to High Performance Scientific Computing*. Lulu.com, 2012.

Hesham EL-REWINI et Mostafa ABD-EL-BARR : *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience, 2005.

Y. FELDMAN, N. DERSHOWITZ et Z. HANNA : Parallel multithreaded satisfiability solver : Design and implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

Hilmar FINNSSON : Generalized monte-carlo tree search extensions for general game playing. *In AAAI*. AAAI Press, 2012.

Michael J. FLYNN : Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21 (9):948–960, septembre 1972. ISSN 0018-9340.

Jon William FREEMAN : *Improvements to Propositional Satisfiability Search Algorithms*. Ph.d. thesis, University of Pennsylvania, Department of Computer and Information Science, 1995a.

Jon William FREEMAN : *Improvements to Propositional Satisfiability Search Algorithms*. Thèse de doctorat, Philadelphia, PA, USA, 1995b. UMI Order No. GAX95-32175.

M. GEBSER, B. KAUFMANN, A. NEUMANN et T. SCHAUB : clasp : A conflict-driven answer set solver. *In* C. BARAL, G. BREWKA et J. SCHLIPF, éditeurs : *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 de *Lecture Notes in Artificial Intelligence*, pages 260–265. Springer-Verlag, 2007.

Sylvain GELLY et Yizao WANG : Exploration exploitation in Go : UCT for Monte-Carlo Go. *In NIPS : Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, décembre 2006.

Ian GENT, Chris JEFFERSON et Ian MIGUEL : Watched literals for constraint propagation in minion. *In* Frédéric BENHAMOU, éditeur : *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 de *Lecture Notes in Computer Science*, pages 182–197. Springer Berlin / Heidelberg, 2006.

JÃ©rÃ´me GIANOLI : Processeur 10 nm, intel donne rendez-vous en fin d'annÃ©e, March 2017. Lien vers l'article.

Allen GOLDBERG : On the complexity of the satisfiability problem. Rapport technique, New York University, 1979.

Eugene GOLDBERG et Yakov NOVIKOV : Berkmin : A fast and robust sat-solver. *Journal of Discrete Applied Mathematics*, 155(12):1549–1561, 2007. ISSN 0166-218X.

Carla P. GOMES, Bart SELMAN, Nuno CRATO et Henry KAUTZ : Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24:67–100, February 2000.

Ryan E. GRANT, Anthony SKJELLUM et Purushotham V.BANGALORE : *Lightweight Threading with MPI Using Persistent Communications Semantics*. United States. National Nuclear Security Administration, 2015.

L. GUO et JM. LAGNIEZ : Dynamic polarity adjustment in a parallel SAT solver. *In ICTAI*, pages 67–73, 2011a.

Long GUO, Youssef HAMADI, Saïd JABBOUR et Lakhdar SAIS : Diversification and intensification in parallel SAT solving. *In CP*, volume 6308 de *Lecture Notes in Computer Science*, pages 252–265. Springer, 2010.

Long GUO, Said JABBOUR et Lakhdar SAIS : Stratégies d'élimination des clauses apprises dans les solveurs sat modernes. 01 2013.

Long GUO et Jean-Marie LAGNIEZ : Dynamic polarity adjustment in a parallel SAT solver. *In ICTAI*, pages 67–73. IEEE Computer Society, 2011b.

John L. GUSTAFSON : Reevaluating amdahl's law. *Journal of Communications of the ACM*, 31(5):532–533, 1988.

Georg HAGER et Gerhard WELLEIN : *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st édition, 2010.

Youssef HAMADI, Saïd JABBOUR, Cédric PIETTE et Lakhdar SAIS : Deterministic parallel DPLL. *JSAT*, 7(4):127–132, 2011.

Youssef HAMADI, Said JABBOUR et Lakhdar SAIS : Control-based clause sharing in parallel sat solving. *In Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 499–504, San Francisco, CA, USA, 2009a. Morgan Kaufmann Publishers Inc.

Youssef HAMADI, Saïd JABBOUR et Lakhdar SAÏS : Learning for dynamic subsumption. *In Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 328–335, Washington, DC, USA, 2009b. IEEE Computer Society.

Youssef HAMADI, Saïd JABBOUR et Lakhdar SAIS : Manysat : a parallel SAT solver. *JSAT*, 6(4):245–262, 2009c.

Hyojung HAN et Fabio SOMENZI : On-the-fly clause improvement. *In Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 209–222, Berlin, Heidelberg, 2009. Springer-Verlag.

Pierre HANSEN, Nenad MLADENOVIĆ et Dionisio PEREZ-BRITOS : Variable neighborhood decomposition search. *Journal of Heuristics*, 7:335–350, 2001.

Jin-Kao HAO et Dorne RAPHAËL : An empirical comparison of two evolutionary methods for satisfiability problems. *In International Conference on Evolutionary Computation (ICEC'94)*, pages 451–455, 1994.

Stephen J. HARTLEY : Steiner systems and the boolean satisfiability problem. *In Proceedings of Symposium on Applied Computing (SAC)*, pages 277–281. ACM, 1996.

John L. HENNESSY et David A. PATTERSON : *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th édition, 2011.

Julien HENRY, Aditya THAKUR, Nicholas KIDD et Thomas REPS : Dissolve : A distributed sat solver based on stalmarck's method.

P. R. HERWIG, M. J. H. HEULE, P. M. van LAMBALGEN et Hans van MAAREN : A new method to construct lower bounds for van der waerden numbers. *Journal of Combinatorics*, 14(1), 2007.

HEULE et van MAAREN : march hi. *SAT competitive events booklet*, 2009.

M. HEULE, O. KULLMANN, S. WIERINGA et A. BIERE : Cube and conquer : Guiding CDCL SAT solvers by lookaheads. *In Proceedings of 7th Haifa Verification Conference (HVC)*, volume 7261 de *LNCS*, pages 50–65. Springer, 2012. **Best paper award**, reported 2011 too, now formally published.

Marijn HEULE, Matti JÄRVISALO et Armin BIERE : Clause elimination procedures for CNF formulas. *In LPAR (Yogyakarta)*, volume 6397 de *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010.

Marijn HEULE, Matti JÄRVISALO et Armin BIERE : Efficient CNF simplification based on binary implication graphs. *In SAT*, volume 6695 de *Lecture Notes in Computer Science*, pages 201–215. Springer, 2011.

Marijn HEULE et Hans van MAAREN : March_dl : Adding adaptive heuristics and a new branching strategy. *JSAT*, (1-4):47–59, 2006a.

Marijn HEULE et Hans van MAAREN : March_dl : Adding adaptive heuristics and a new branching strategy. *JSAT*, (1-4):47–59, 2006b.

Marijn J. H. HEULE, Oliver KULLMANN et Victor W. MAREK : Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. *In Proceedings of the Nineteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 de *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

John N. HOOKER et V. VINAY : Branching rules for satisfiability (extended abstract). *In Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 426–437, London, UK, 1994. Springer-Verlag. ISBN 3-540-58715-2.

Jinbo HUANG : The effect of restarts on the efficiency of clause learning. *In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)* pro (2007), pages 2318–2323.

A. HYVÄRINEN, T. JUNTTILA et I. NIEMELÄ : Partitioning SAT instances for distributed solving. *In LPAR*, pages 372–386, 2010a.

Antti Eero Johannes HYVÄRINEN, Tommi A. JUNTTILA et Ilkka NIEMELÄ : Partitioning SAT instances for distributed solving. *In LPAR (Yogyakarta)*, volume 6397 de *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010b.

Antti Eero Johannes HYVÄRINEN, Tommi A. JUNTTILA et Ilkka NIEMELÄ : Grid-based SAT solving with iterative partitioning and clause learning. *In CP*, volume 6876 de *Lecture Notes in Computer Science*, pages 385–399. Springer, 2011.

Saïd JABBOUR, Jerry LONLAC et Lakhdar SAÏS : Adding new bi-asserting clauses for faster search in modern SAT solvers. *In SARA*. AAAI, 2013.

Joseph JAJA : *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

Matti JARVISALO, Armin BIERE et Marijn HEULE : Blocked clause elimination. *In* Javier ESPARZA et Rupak MAJUMDAR, éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 de *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin / Heidelberg, 2010.

James JEFFERS et James REINDERS : *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st édition, 2013.

Robert G. JEROSLOW et Jinchang WANG : Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

D.S. JOHNSON et M.A. TRICK, éditeurs. *Selected papers of Second DIMACS Challenge on Satisfiability Testing organized by the Center for Discrete Mathematics and Computer Science of Rutgers University*, volume 26 de *DIMACS Series in Discrete Mathematics and Theorical Computer Science, American Mathematical Society*, 1996.

Bernard JURKOWIAK, Chu Min LI et Gil UTARD : Parallelizing satz using dynamic workload balancing. *In In Proceedings of Workshop on Theory and Applications of Satisfiability Testing (SAT'2001*, pages 205–211. Elsevier Science Publishers, 2001.

Henry KAUTZ et Bart SELMAN : Pushing the envelope : planning, propositional logic, and stochastic search. *In Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, AAAI'96, pages 1194–1201. AAAI Press, 1996.

Fukazawa KEIICHIRO et Nanri TAKESHI : Performance of large scale MHD simulation of global planetary magnetosphere with massively parallel scalar type supercomputer including post processing. *In Proceedings of the Fourteenth International Conference on High Performance Computing and Communication (HPCC) & the Ninth International Conference on Embedded Software and Systems (ICESS)*, pages 976–982, Liverpool, United Kingdom, 2012. Computer Society.

Donald E. KNUTH : *The Art of Computer Programming, Volume 3 : (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.

Levente KOCSIS et Csaba SZEPESVÁRI : Bandit based monte-carlo planning. pages 282–293, 2006.

Boris KONEV et Alexei LISITSA : A SAT attack on the erdős discrepancy conjecture. *In Theory and Applications of Satisfiability Testing SAT'14, 17th International Conference, Proceedings*, pages 219–226, 2014.

Ajay D. KSHEMKALYANI et Mukesh SINGHAL : Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering TSE*, 20(1):43–54, 1994.

Oliver KULLMANN : Investigations on autark assignments. *Discrete Applied Mathematics*, 107:2000, 1998.

S. R. KUMAR, A. RUSSELL et R. SUNDARAM : Approximating latin square extensions. *Algorithmica*, 24(2):128–138, Jun 1999.

Jean-Marie LAGNIEZ, Daniel Le BERRE, Tiago de LIMA et Valentin MONTMIRAIL : A recursive shortcut for CEGAR : application to the modal logic K satisfiability problem. *In IJCAI*, pages 674–680. ijcai.org, 2017.

Nadjib LAZAAR, Youssef HAMADI, Said JABBOUR et Michèle SEBAG : BESS : Bandit Ensemble for parallel SAT Solving. Research report, septembre 2012.

Harry R. LEWIS : Renaming a set of clauses as a horn set. *Journal of ACM*, 25(1):134–135, janvier 1978.

Chu Min LI et Anbulagan ANBULAGAN : Heuristics based on unit propagation for satisfiability problems. *In Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1*, pages 366–371, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

Jia Hui LIANG, Vijay GANESH, Pascal POUPART et Krzysztof CZARNECKI : Exponential recency weighted average branching heuristic for SAT solvers. *In* Dale SCHUURMANS et Michael P. WELLMAN, éditeurs : *Proceedings of the Thirtieth Conference on Artificial Intelligence (AAAI)*, pages 3434–3440. AAAI Press, 2016a.

Jia Hui LIANG, Vijay GANESH, Pascal POUPART et Krzysztof CZARNECKI : Learning rate based branching heuristic for SAT solvers. *In Proceedings of the Nineteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 de *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016b.

Paolo LIBERATORE : On the complexity of choosing the branching literal in dpll. *Artificial Intelligence*, 116:315–326, January 2000. ISSN 0004-3702.

Sandeep LODHA et Ajay D. KSHEMKALYANI : A fair distributed mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 11(6):537–549, 2000.

Michael LUBY, Alistair SINCLAIR et David ZUCKERMAN : Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993. ISSN 0020-0190.

Mao LUO, Chu-Min LI, Fan XIAO, Felip MANYA et Zhipeng LU : An effective learnt clause minimization approach for cdcl sat solvers. *In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 703–711, 2017.

Yogesh S. MAHAJAN, Zhaohui FU et Sharad MALIK : Zchaff2004 : An efficient sat solver. *In Proceedings of the the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 360–375, Berlin, Heidelberg, 2005. Springer-Verlag.

Michal MALAFIEJSKI, Krzysztof GIARO, Robert JANCZEWSKI et Marek KUBALE : Sum coloring of bipartite graphs with bounded degree. *Algorithmica*, 40(4):235–244, 2004.

Norbert MANTHEY, Tobias PHILIPP et Christoph WERNHARD : Soundness of inprocessing in clause sharing sat solvers. *In Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, SAT'13, pages 22–39, Berlin, Heidelberg, 2013. Springer-Verlag.

Iser MARKUS, Sinz CARSTEN et Taghdiri MANA : Minimizing models for tseitin-encoded SAT instances. *In Proceedings of the Sixteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7962 de *Lecture Notes in Computer Science*, pages 224–232. Springer, 2013.

Joao MARQUES-SILVA et Ines LYNCE : On improving mus extraction algorithms. *In Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 159–173, Berlin, Heidelberg, 2011. Springer-Verlag.

João P. MARQUES-SILVA et Karem A. SAKALLAH : Grasp—a new search algorithm for satisfiability. *In ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7.

R. MARTINS, V. M. MANQUINHO et I. LYNCE : Improving search space splitting for parallel SAT solving. *In ICTAI*, pages 336–343, 2010.

Kulkarni MILIND, Burtscher MARTIN, Inkulu RAJASEKHAR, Pingali KESHAV et Cascaval CALIN : How much parallelism is there in irregular applications ? *In Proceedings of the Fourteenth Symposium on Principles and Practice Of Parallel Programming (PPOPP)*, pages 3–14, Raleigh, NC, USA, 2009. ACM.

Michel MINOUX : LTUR : A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29(1):1–12, 15 septembre 1988.

David MITCHELL, Bart SELMAN et Hector J. LEVESQUE : Hard and easy distributions of sat problems. *In Proceedings of the Tenth American National Conference on Artificial Intelligence (AAAI'92)*, pages 459–465, 1992.

Burkhard MONIEN et Ewald SPECKENMEYER : Solving satisfiability in less than 2^n steps. *Journal of Discrete Applied Mathematics*, 10:287–295, 1985.

Gordon E. MOORE : Technical digest. international electron devices meeting. *Journal of Solid-State Circuits Society Newsletter*, pages 11–13, 1975. Lien vers l'article.

Matthew W. MOSKEWICZ, Conor F. MADIGAN, Ying ZHAO, Lintao ZHANG et Sharad MALIK : Chaff : engineering an efficient sat solver. *In DAC '01 : Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, juin 2001a. ACM. ISBN 1-58113-297-2.

Matthew W. MOSKEWICZ, Conor F. MADIGAN, Ying ZHAO, Lintao ZHANG et Sharad MALIK : Chaff : Engineering an efficient sat solver. *In Proceedings of the 38th Annual Design Automation Conference (DAC)*, DAC '01, pages 530–535, New York, NY, USA, 2001b.

MPICH2. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

A. NADEL, M. GORDON, A. PATTI et Z. HANNA : Eureka-2006 sat solver solver description for sat-race 2006, 2006.

Alexander NADEL et Vadim RYVCHIN : Assignment stack shrinking. *In* Ofer STRICHMAN et Stefan SZEIDER, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2010*, volume 6175 de *Lecture Notes in Computer Science*, pages 375–381. Springer Berlin / Heidelberg, 2010.

NATIONAL SCIENCE FOUNDATION : *Advisory Committee for Cyberinfrastructure. Task Force on Grand Challenges*. Task Force, 2011. Lien vers l'article.

Saeed NEJATI, Zack NEWSHAM, Joseph SCOTT, Jia Hui LIANG, Catherine GEBOTYS, Pascal POUPART et Vijay GANESH : A propagation rate based splitting heuristic for divide-and-conquer solvers. *In International Conference on Theory and Applications of Satisfiability Testing*, pages 251–260. Springer, 2017.

Shojiro NISHIO, Kin F. LI et Eric G. MANNING : A resilient mutual exclusion algorithm for computer networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 1(3):344–356, 1990.

Chanseok OH : Between SAT and UNSAT : the fundamental difference in CDCL SAT. *In Proceedings of the Eighteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 de *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015.

Kei OHMURA et Kazunori UEDA : c-sat : A parallel SAT solver for clusters. *In Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 de *Lecture Notes in Computer Science*, pages 524–537. Springer, 2009.

Peng Sing OW et Thomas E. MORTON : Filtered beam search in scheduling. *International Journal of Production Research*, 26:35–62, 1988.

Anastasia PAPARRIZOU et Kostas STERGIOU : Evaluating simple fully automated heuristics for adaptive constraint propagation. *In ICTAI*, pages 880–885. IEEE Computer Society, 2012.

Behrooz PARHAMI : *Introduction to Parallel Processing : Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

Cédric PIETTE, Youssef HAMADI et Saïs LAKHDAR : Vivifying propositional clausal formulae. *In Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'03)*, pages 525–529, Patras (Greece), juillet 2008.

Knot PIPATSRISAWAT et Adnan DARWICHE : A lightweight component caching scheme for satisfiability solvers. *In Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 294–299, 2007.

Knot PIPATSRISAWAT et Adnan DARWICHE : Width-based restart policies for clause-learning satisfiability solvers. *In Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 341–355, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02776-5.

David A. PLAISTED et Steven GREENBAUM : A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, septembre 1986a.

David A. PLAISTED et Steven GREENBAUM : A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, septembre 1986b. ISSN 0747-7171.

Daniele PRETOLANI : *Satisfiability and Hypergraphs*. Thèse de doctorat, dipartimento di Informatica : Universita di Pisa, Genova, Italia, mars 1993. TD-12/93.

Sanguthevar RAJASEKARAN et John REIF : *Handbook of Parallel Computing : Models, Algorithms and Applications*. Chapman & Hall/Crc Computer & Information Science Series, 2007.

Antoine RAUZY : Polynomial restrictions of sat : What can be done with an efficient implementation of the davis and putnam's procedure. *In* pro (1995), pages 515–532.

Herbert ROBBINS : Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.*, 58 (5):527–535, 1952.

S. Sinkovits ROBERT et G. Duda MICHAEL : Optimization and parallel load balancing of the MPAS atmosphere weather and climate code. *In Proceedings of of the Extreme Science and Engineering Discovery Environment (XSEDE)*, pages 5 :1–5 :6, Miami, USA, 2016. ACM.

John Alan ROBINSON : A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

Olivier ROUSSEL : Description of ppfolio. Rapport technique, 2011.

Lawrence RYAN : Efficient algorithms for clause-learning sat solvers, 2004.

SAT. SAT-race, 2015. http://baldur.iti.kit.edu/sat-race-2015/.

T. SCHUBERT, M. LEWIS et B. BECKER : Pamiraxt : Parallel SAT solving with threads and message passing. *JSAT*, 6(4):203–222, 2009.

Sven SCHULZ et Wolfgang BLOCHINGER : Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing*, 8(3):443–471, 2010.

Bart SELMAN, Henry A. KAUTZ et Bram COHEN : Noise strategies for improving local search. *In Proceedings of the Twelfth American National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.

Bart SELMAN, Hector J. LEVESQUE et David MITCHELL : Gsat : A new method for solving hard satisfiability problems. *In Proceedings of the Tenth American National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

A. SEMENOV et O. ZAIKIN : Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. *In PaCT*, pages 222–230, 2015.

SENSEIS : Jeu qui implémente uct : http ://senseis.xmp.net/ ?uct. 2014.

Claude Elwood SHANNON : A symbolic analysis of relay and switching circuits. Thesis (m.s.), Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1940.

João P. Marques SILVA : The impact of branching heuristics in propositional satisfiability algorithms. *In Proceedings of the 9th Portuguese Conference on Artificial Intelligence : Progress in Artificial Intelligence*, EPIA '99, pages 62–74, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66548-X.

Sievers SILVAN : Implementation of the uct algorithm for dopp elkopf. 2012.

Mukesh SINGHAL : A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993.

Carsten SINZ, Wolfgang BLOCHINGER et Wolfgang KÜCHLIN : Pasat - parallel sat-checking with lemma exchange : Implementation and applications. *In LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001), volume 9 of Electronic Notes in Discrete Mathematics*. Elsevier Science Publishers, 2001.

Tomohiro SONOBE, Shuya KONDOH et Mary INABA : Community branching for parallel portfolio SAT solvers. *In Proceedings of the Seventeenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 de *Lecture Notes in Computer Science*, pages 188–196. Springer, 2014.

Mate SOOS : Cryptominisat 2.5. 0. *SAT Race competitive event booklet*, 2010.

Niklas SÖRENSSON et Armin BIERE : Minimizing learned clauses. *In Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

Niklas SÖRENSSON et Niklas EÉN : MiniSat 2.1 and MiniSat++ 1.0 - SAT Race 2008 Editions. Rapport technique, 2008.

Niklas SÖRENSSON et Niklas EÉN : Minisat 2.1 and minisat++ 1.0 - sat race 2008 editions. *SAT 2009 competitive events booklet : preliminary version*, page 31, 2009.

Sathiamoorthy SUBBARAYAN et Dhiraj K. PRADHAN : NiVER : Non increasing variable elimination resolution for preprocessing sat instances. *In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)* pro (2005b), pages 276–291.

Xian-He SUN et Lionel M. NI : Another view on parallel speedup. *In Proceedings of the Conference on Supercomputing*, Supercomputing, pages 324–333, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

Andrew TANENBAUM : *Architecture de l'ordinateur*. Pearson Education, 2005.

Rajeev THAKUR et William GROPP : Test suite for evaluating performance of MPI implementations that support mpi_thread_multiple. *In Parallel Virtual Machine PVM/MPI*, volume 4757, pages 46–55. Springer, 2007.

G.S. TSEITIN : On the complexity of derivations in the propositional calculus. *In* H.A.O. SLESENKO, éditeur : *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

Moschini UGO, Teeninga PAUL, C. Trager SCOTT et H. F. Wilkinson MICHAEL : Parallel 2d local pattern spectra of invariant moments for galaxy classification. *In Proceedings of the Sixteenth International Conference on Computer Analysis of Images and Patterns (CAIP)*, volume 9257 de *Lecture Notes in Computer Science*, pages 121–133, Valletta, Malta, 2015. Springer.

P. van der TAK, M. HEULE et A. BIERE : Concurrent cube-and-conquer. *CoRR*, abs/1402.4465, 2014.

Pascal VANDER-SWALMEN, Gilles DEQUEN et Michaël KRAJECKI : On multi-threaded satisfiability solving with openmp. *In Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 146–157, Berlin, Heidelberg, 2008. Springer-Verlag.

Pascal VANDER-SWALMEN, Gilles DEQUEN et Michaël KRAJECKI : A collaborative approach for multi-threaded sat solving. *Int. J. Parallel Program.*, 37(3):324–342, juin 2009.

Wanxia WEI et Chu Min LI : Switching between two adaptive noise mechanisms in local search for sat. *SAT 2009 competitive events booklet : preliminary version*, page 57, 2009.

Siert WIERINGA et Keijo HELJANKO : Concurrent clause strengthening. *In Proceedings of the Sixteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7962 de *Lecture Notes in Computer Science*, pages 116–132. Springer, 2013.

Weigang WU, Jiebin ZHANG, Aoxue LUO et Jiannong CAO : Distributed mutual exclusion algorithms for intersection traffic control. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26 (1):65–74, 2015.

Lin XU, Holger HOOS et Kevin LEYTON-BROWN : Hydra : Automatically configuring algorithms for portfolio-based selection. *In AAAI*. AAAI Press, 2010.

T YASUMOTO et T OKUGAWA : Rokk. *SAT Competition*, 2014.

Hantao ZHANG : Sato : An efficient prepositional prover. *In* William MCCUNE, éditeur : *Automated Deduction–CADE-14*, volume 1249 de *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin / Heidelberg, 1997.

Hantao ZHANG et Maria Paola BONACINA : Cumulating search in a distributed computing environment : A case study in parallel satisfiability. *In Proceedings of the First Intelligence Symposium on Parallel Symbolic Computation*, pages 422–431. Publishing Company, 1994.

Hantao ZHANG, Maria Paola BONACINA et Jieh HSIANG : Psato : A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, juin 1996.

Lintao ZHANG, Conor F. MADIGAN, Matthew H. MOSKEWICZ et Sharad MALIK : Efficient conflict driven learning in a boolean satisfiability solver. *In ICCAD '01 : Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, novembre 2001a. IEEE Press. ISBN 0-7803-7249-2.

Lintao ZHANG, Conor F. MADIGAN, Matthew W. MOSKEWICZ et Sharad MALIK : Efficient conflict driven learning in boolean satisfiability solver. *In Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE Computer Society, 2001b.

Résumé

La thèse porte sur la résolution des problèmes de satisfaisabilité booléenne (SAT) dans un cadre massivement parallèle. Le problème SAT est largement utilisé pour résoudre des problèmes combinatoires de première importance comme la vérification formelle de matériels et de logiciels, la bio-informatique, la cryptographie, la planification et l'ordonnancement de tâches. Plusieurs contributions sont apportées dans cette thèse. Elles vont de la conception d'algorithmes basés sur les approches « portfolio » et « diviser pour mieux régner », à l'adaptation de modèles de programmation parallèle, notamment hybride (destinés à des architectures à mémoire partagée et distribuée), à SAT, en passant par l'amélioration des stratégies de résolution. Ce travail de thèse a donné lieu à plusieurs contributions dans des conférences internationales du domaine ainsi qu'à plusieurs outils (open sources) de résolution des problèmes SAT, compétitifs au niveau international.

Mots-clés: SAT, solveurs SAT, parallèle, calcul distribué, modèles de programmation hybride

Abstract

This thesis deals with propositional satisfiability (SAT) in a massively parallel setting. The SAT problem is widely used for solving several combinatorial problems (e.g. formal verification of hardware and software, bioinformatics, cryptography, planning, scheduling, etc.). The first contribution of this thesis concerns the design of efficient algorithms based on the approaches « portfolio » and « divide and conquer ». Secondly, an adaptation of several parallel programming models including hybrid (parallel and distributed computing) to SAT is proposed. This work has led to several contributions to international conferences and highly competitive distributed SAT solvers.

Keywords: SAT, SAT solvers, parallel, distributed computing, hybrid programming models