Aspects parallèles des problèmes de satisfaisabilité

Pascal Vander-Swalmen

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE
en collaboration avec
l'Université de Picardie Jules Verne

UFR Sciences Exactes et Naturelles (URCA), UFR des Sciences (UPJV)
CReSTIC (URCA), MIS (UPJV)

École Doctorale Sciences, Technologies, Santé

THÈSE

pour obtenir le grade de

Docteur de l'Université de Reims Champagne-Ardenne
en Informatique

par
Pascal VANDER-SWALMEN

le 7 décembre 2009

Aspects parallèles des problèmes de satisfaisabilité

JURY

M. :	William	JALBY	Président
MM. :	Ivan	LAVALLÉE	Rapporteurs
	Lakhdar	SAÏS	
MM. :	Laure	DEVENDEVILLE	Examinateurs
	Daniel	SINGER	
MM. :	Michaël	KRAJECKI	Directeurs
	Gilles	DEQUEN	

Le chemin est long mais la voie est libre

Remerciements

Je tiens à remercier monsieur Ivan Lavallée, scientifique connu et reconnu qui a accepté de rapporter ma thèse. J'ai été heureux d'apprendre qu'il partageait mon intérêt pour le problème de satisfaisabilité. De plus, ses travaux dans le domaine du parallélisme ont été très importants et pionniers.

Je remercie également monsieur Lakhdar Saïs, spécialiste des problèmes de satisfaction de contraintes, d'avoir bien voulu rapporter cette thèse. Depuis quelques années il s'attaque à ce problème sous l'angle du parallélisme avec succès, et je suis fier qu'il ait pu donner son jugement sur ce travail.

Monsieur William Jalby me fait l'honneur d'être présent au sein de mon jury afin d'examiner ma thèse. Ses connaissances dans les machines parallèles sont un plus très appréciable pour valider ce travail.

Merci à Madame Laure Devendeville d'avoir accepté le rôle d'examinatrice de ma thèse. Tout au long de mes études, elle a été présente pour répondre à toutes mes questions, que ce soit en tant qu'étudiant ou de l'autre côté du bureau.

Monsieur Daniel Singer, qui joue un rôle indéniable dans la résolution de SAT en parallèle (par exemple en organisant des sessions de travail) a accepté de faire partie de mon jury, et je l'en remercie.

Les deux derniers représentants de mon jury, et non des moindres, sont mes directeurs de thèse : monsieur Michaël Krajecki et monsieur Gilles Dequen. Je remercie tout d'abord Michaël Krajecki qui m'a permis d'effectuer une thèse d'Informatique alors qu'il ne me connaissait pas. Il m'a donc fait confiance dès le départ de cette aventure, cela me touche et j'espère qu'il ne l'a jamais regretté. Ses conseils éclairés concernant les aspects parallèles ont toujours été très intéressants et ont toujours porté leurs fruits. Malgré une charge de travail importante due à ses fonctions de Vice-Président TICE de l'URCA, il a toujours tout fait pour être présent au moins par téléphone dans les moments les plus difficiles et les plus tendus. Il a également su me guider dans les méandres administratifs de l'Université sans perdre son sang froid. Je salue tous les efforts qu'il fournit pour donner à ses équipes les moyens de travailler, le calculateur ROMEO II en est, à mon sens, le meilleur exemple car il permet de travailler à proprement parler mais aussi de faire travailler. Le co-directeur de ma thèse, Gilles Dequen, m'a fait l'honneur de proposer mon nom à Michaël lorsqu'ils avaient ce projet en tête. Je le remercie de la confiance qu'il m'a témoignée en me rappelant deux ans après un stage de DEA effectué au sein de son équipe. À l'époque, cet appel téléphonique providentiel m'avait beaucoup ému, je savais que cette aventure serait difficile mais intéressante.

Gilles donne toujours de bons conseils et est perspicace quand il s'agit de réduire un arbre de recherche ou pour programmer efficacement. Présent et prêt à aider même les week-ends, il m'a toujours soutenu pendant ces *quasi* trois années de recherche. Dans le domaine de l'enseignement, Gilles a été une sorte de tuteur, il m'a très souvent aidé pour les tâches liées à cette fonction. Ces deux hommes m'ont donné beaucoup, chacun dans son domaine de recherche de prédilection a su guider cette thèse pour qu'elle devienne ce qu'elle est aujourd'hui. Pour tout cela je leur dis un grand merci. Toutefois, j'irai encore un peu plus loin car outre des qualités de chercheurs, ils ont tous deux des qualités humaines qui m'ont porté tout au long de cette thèse : écoute, disponibilité, respect, motivation et sympathie. Ce cocktail fait que, même si ce métier demande parfois de lourds sacrifices en termes de temps de travail et pour la vie de famille, ces sacrifices sont moins difficiles à faire. Enfin, je dirai que mes meilleurs souvenirs de thèse sont les réunions que nous faisions ensemble car je sentais qu'à ces moments-là se jouaient des choses vraiment importantes, que de réelles avancées étaient en train de se produire, le tout dans la bonne humeur et le respect de chacun. Tout cela est très important pour moi, j'ai deux bons modèles d'enseignant-chercheur à suivre.

Afin de mener une thèse dans de bonnes conditions, il est primordial d'avoir un environnement de travail adéquat. J'ai trouvé cela dans mes deux équipes de recherche. Étant domicilié à Amiens, j'ai moins côtoyé les membres du CReSTIC (Reims), mais ce n'est pas pour autant que je ne suis pas bien accueilli à chacune de mes visites. Je remercie tous les membres du laboratoire pour leur bonne humeur et leur aide précieuse sur d'innombrables sujets. En particulier, je remercie Hervé Deleau pour sa patience infinie lorsque je débutais dans une réelle utilisation des systèmes GNU/Linux (environ 3 mois de questions sans interruption) et pour tous les services qu'il m'a rendu par la suite. Je remercie tous les membres du MIS (Amiens) pour leur bonne humeur également. J'ai apprécié les échanges que nous avons eus au sujet de la recherche mais aussi de l'enseignement. Je souhaite remercier en particulier les doctorants du MIS avec qui il est toujours passionnant de discuter longuement concernant des sujets terre-à-terre ou que je rapprocherais parfois de la philosophie. Après tout, une thèse est une *Ph D*, et ces échanges forment l'homme qui ensuite réfléchit à son sujet de thèse avec, peut-être, un autre regard. Je remercie en particulier Sylvain Darras avec qui j'ai partagé mes années d'études et mon bureau de doctorant pendant plusieurs années. Il m'a beaucoup aidé sur différents points tout au long de la thèse. Il me semble évident et nécessaire de remercier également tous les personnels travaillant dans les laboratoires. Sans eux, les couloirs ne seraient pas propres, les tâches administratives seraient encore plus compliquées et nous ne serions pas reliés à Internet... Autant dire que ces points sont primordiaux pour mener un travail de qualité.

Je tiens à remercier l'ensemble de la communauté des logiciels libres : des programmeurs aux traducteurs en passant par les graphistes. Sans ces personnes qui prennent le temps d'en donner aux autres afin de rendre l'outil informatique accessible au maximum de gens, je n'aurai pas pu travailler dans les conditions qui m'ont semblé adéquates et nécessaires. J'espère un jour rejoindre de façon significative cette communauté.

Merci à mes amis avec qui j'ai pu de temps en temps sortir du contexte de la thèse afin de faire le vide. Que ce soit autour d'une table ou sur un tatami, ce vide est

appréciable car il permet de prendre du recul et revenir plus serein au travail.

Merci à ma famille (à savoir que je considère ma belle-famille comme ma famille) pour le soutien sans limite qu'elle m'a apporté. Ce soutien a pu prendre différentes formes, mais à chaque fois ce fut un grand soulagement lors de son apparition. Merci à Céline pour son soutien, la vision que nous partageons et surtout pour m'avoir apporté Flore qui est le meilleur antidépresseur que je connaisse et mon nouveau moteur.

Je dédis ce travail à mon grand-frère, Sébastien.

Résumé

Malgré sa complexité de résolution, le problème de SATisfaisabilité est une excellente et compétitive approche pour résoudre un large éventail de problèmes. Cela génère une forte demande pour une résolution de SAT haute performance de la part des industriels. Au fil du temps, de nombreuses approches et optimisations différentes ont été développées pour résoudre le problème plus efficacement. Ces innovations ont été faites sans prendre en compte le développement des micro processeurs actuels qui voient le nombre de leur cœurs de calcul augmenter. Cette thèse présente un nouveau type d'algorithme parallèle basé sur une forte collaboration où un processus *riche* est en charge de l'évaluation de l'arbre de recherche et où des processus *pauvres* fournissent des informations partielles ou globales, heuristiques ou logiques afin de simplifier la tâche du riche. Pour concrétiser ce solveur et le rendre efficace, nous avons étendu la notion de *chemin de guidage* à celle d'*arbre de guidage*. L'arbre de recherche est totalement partagé en mémoire centrale et tous les processeurs peuvent y travailler en même temps. Ce nouveau solveur est appelé MTSS pour *Multi-Threaded* SAT *Solver*. De plus, nous avons implémenté une tâche pour les processus riche et pauvres qui leur permet d'exécuter un solveur SAT externe, et cela, avec ou sans échange de lemmes afin de paralléliser tous types de solveurs (dédiés aux formules industrielles ou aléatoires). Ce nouvel environnement facilite la parallélisation des futures implémentations pour SAT. Quelques exemples et expérimentations, avec ou sans échange de lemmes, de parallélisation de solveurs externes sont présentées, mais aussi des résultats sur les performances de MTSS. Il est intéressant de noter que certaines accélérations sont super linéaires.

Mots-clefs : optimisation combinatoire, satisfaisabilité, parallélisme, architecture multi-cœurs.

Abstract

In spite of its computational complexity, the SATisfiability problem is a great and competitive approach to solve a wide range of problems. This leads to have a strong demand for high-performance SAT-solving tools in industry. Over the years, many different approaches and optimizations have been developed to tackle the problem more efficiently while being unaware of the actual trend in processor development which is from single-core to multi-core CPUs. This thesis presents a new kind of parallel algorithm with a strong collaborative approach where a *rich* thread is in charge of the search-tree evaluation and where a set of *poor* threads yield partial or global, logical or heuristics information to simplify the rich task. To become this solver real and efficient, we extended the *guiding path* notion to the *guiding tree* notion. The entire search-tree is shared in the main memory and each processor can work on it at the same time. This new solver is named MTSS for *Multi-Threaded* SAT *Solver*. In addition, we implemented a task for rich and poor threads which can execute an external SAT solver with or without lemma exchange policy. This permits to parallelize all kind of solvers (dedicated to industrial or random formulas). This new framework facilitates the future parallel SAT solving implementation approaches. Some examples and experimentations with and without lemma exchange strategies parallelizing external solvers are presented, as well as some benchmarks on MTSS itself. Sometimes, speedups are super linear.

Keywords : Combinatorial Optimization, Satisfiability, Parallelism, Multi-core Architecture.

Table des matières

Introduction

L'Homme a toujours construit des outils pour faciliter sa vie. Aujourd'hui nous sommes à l'ère de l'information et l'ordinateur en est l'outil de base, il a profondément changé notre société alors que c'est une invention très récente. L'Informatique, qui se définit comme le traitement automatique de l'information est une science qui connaît un essor récent comparé à celui d'autres sciences et surtout très important ces dernières décennies. Ce récent essor est dû à l'invention des ordinateurs. Grâce à lui, les problèmes décrits et résolus par l'algorithmique (une branche de l'Informatique et des Mathématiques) ont pu être implémentés afin de les faire exécuter par la machine plutôt que par l'Homme. L'algorithmique traite des problèmes décidables par étapes successives. Trouver les solutions de problèmes difficiles peut donc prendre un temps beaucoup trop long pour l'être humain, là où un ordinateur peut très rapidement donner la solution. Toutefois, il existe des problèmes dont la terminaison n'est pas assurée dans un temps acceptable même par un ordinateur très récent. Une catégorie de ces problèmes est celle des problèmes combinatoires. Ils relèvent de la Combinatoire et regroupent les problèmes pour lesquels le nombre de possibilités explose en fonction de la taille de l'instance du problème. Les algorithmes qui résolvent les problèmes combinatoires de décision doivent parcourir l'ensemble des interprétations possibles afin d'y trouver, éventuellement, une solution. Il existe aussi les problèmes d'optimisation associés, ils consistent à trouver la meilleure des solutions, ou la moins mauvaise. Quelque soit l'ordinateur utilisé, ces algorithmes ont le même objectif : donner une réponse. Seule la manière de résoudre ces problèmes peut différer entre deux algorithmes. Toutefois, il existe une distinction importante entre les algorithmes séquentiels et les algorithmes parallèles. L'algorithmique parallèle permet de résoudre un problème en le subdivisant et en faisant travailler ensemble plusieurs entités. Ce type d'algorithme peut être exécuté sur n'importe quel ordinateur mais a très peu d'intérêts si l'ordinateur n'est pas constitué de plusieurs unités de calcul. Il y a donc des liens entre l'aspect théorique de la programmation et la réalité des ordinateurs disponibles.

Les ordinateurs ont connu beaucoup de révolutions depuis leur invention au milieu du $19^{ème}$ siècle. Ils étaient tout d'abord des machines imposantes, chères, réservées aux militaires et aux scientifiques. Puis la micro-informatique a amené cette technologie partout, jusqu'à aujourd'hui dans nos maisons, nos véhicules, et même dans nos poches. En effet, de nos jours les téléphones mobiles sont plusieurs milliers de fois plus puissants que les premiers ordinateurs, tout en consommant beaucoup moins d'énergie et pour un poids et une taille qui n'ont plus rien à voir. L'évolution est telle qu'elle ap-

proche maintenant certaines limites physiques des matériaux utilisés (en l'occurrence, la chaleur dégagée trop importante). En réponse à cette limite, nos micro-processeurs ont vu l'arrivée massive des processeurs multi-cœurs. Ces puces de silicium embarquent plusieurs cœurs de calcul capables de travailler en parallèle les uns des autres. Il est dorénavant possible, à moindre coût, de profiter de l'algorithmique parallèle.

Cette thèse a pour sujet les aspects parallèles des problèmes de satisfaisabilité. Ce problème, aussi appelé tout simplement SAT, est un problème combinatoire. Les variables du problème sont binaires et les formules à résoudre sont des formules issues de la logique propositionnelle. SAT est un problème de décision. Même si le parallélisme ne réduit pas la complexité du problème SAT, les aspects fondamentaux et industriels ont un tel impact qu'il est naturel de vouloir concevoir des algorithmes parallèles pour SAT lorsque l'algorithmique est poussée au parallélisme par le changement de technologie actuel. Cela permettra de tirer le maximum de performance des technologies les plus récentes et de celles à venir dans un domaine très concurrentiel et très étudié. Ce sont ces constatations qui nous ont conduit à réaliser les travaux qui vont vous être présentés.

Le chapitre 1 décrit les aspects de ce problème. Les méthodes complètes qui correspondent à l'exploration d'un arbre binaire de recherche, basés sur la procédure DLL et les méthodes incomplètes qui permettent de donner une solution rapidement mais sans explorer la totalité de l'espace de recherche (`Walksat` par exemple) sont décrites. Quelques algorithmes correspondant à ces méthodes seront donnés. Les aspects combinatoires et le phénomène de seuil de difficulté des formules SAT générées aléatoirement sont expliqués, ainsi que les nombreuses applications industrielles que ce problème offre. D'ailleurs, il sera aussi discuté des solveurs (les programmes qui résolvent SAT) ayant de bonnes aptitudes sur les formules aléatoires et ceux efficaces sur les formules industrielles. Ces deux types de solveurs sont différents car les formules n'ont pas du tout le même profil. Ainsi, il sera fait référence aux heuristiques de branchement, aux traitements locaux, à l'apprentissage de lemmes, aux politiques de redémarrage, *etc.* Tous les points techniques et algorithmes donnés dans ce chapitre le sont dans un contexte séquentiel.

Le chapitre 2 présentera les motivations qui nous ont poussés à travailler sur le problème SAT dans un contexte parallèle. En effet, à travers une explication de l'augmentation de la puissance des processeurs, nous montrerons qu'il est inéluctable et écologiquement intéressant de produire aujourd'hui des processeurs multi-cœurs. Il est de notre intérêt de savoir utiliser la puissance qu'ils offrent puisque ce type de processeur devient aujourd'hui le standard de l'industrie. Dans ce chapitre, une présentation des différents modèles de parallélisme sera faite. Des modèles d'exécution (SISD, MISD, SIMD et MIMD) et d'architecture (CC-UMA, CC-NUMA, NoRMA) aux modèles de programmation (PRAM et DRAM), ces points seront détaillés. Nous expliquerons que les processeurs multi-cœurs offrent une architecture à mémoire à accès uniforme (MIMD, CC-UMA), et par conséquent, ils peuvent exécuter un modèle utilisant plusieurs instructions et plusieurs données dans le même temps. Le chercheur qui veut écrire un algorithme parallèle à destination des processeurs multi-cœurs peut alors le faire dans un cadre de mémoire partagée (PRAM), un modèle naturel et efficace.

Le chapitre 3 fera le lien entre les modèles donnés au chapitre 2 et le problème décrit

au chapitre 1. Il est possible de résoudre SAT de manière collaborative ou concurrentielle, les problématiques rencontrées ne sont pas les mêmes. Nous aborderons les difficultés à dégager du parallélisme pendant la résolution de SAT pour les approches collaboratives dûes à l'irrégularité des accès aux données et à la dépendance des traitements. Pour concevoir un algorithme parallèle collaboratif efficace, il est important de réduire les instants où les unités de calcul sont oisives ; c'est l'équilibrage de charge. Nous verrons qu'une manière simple et pertinente d'équilibrer la charge entre les unités de calcul pour SAT est d'utiliser le principe de chemin de guidage. Nous définirons cette notion en détails, mais pour résumer, le principe est de faire travailler une entité de calcul sur un sous-arbre identifié par un chemin, lui-même extrait d'un chemin d'exécution d'une autre entité après avoir identifié les sous-arbres non développés. Concernant les approches concurrentes, tous les solveurs ne sont pas exploitables, ils doivent présenter des performances séquentielles assez hétérogènes pour amener un réel intérêt dans une approche concurrente. Par exemple, les solveurs ayant recours à de l'aléatoire ou ceux très sensibles au paramétrage démontrent d'une aptitude à la parallélisation concurrente. Que ce soit de manière collaborative ou concurrente, les solveurs industriels ont la capacité d'apprendre des lemmes pendant leur résolution, il est intéressant de les échanger, mais cela est une nouvelle fois, une source de problèmes que nous détaillerons. Nous parlerons du fait que le parallélisme pour le problème SAT peut jouer un excellent rôle puisqu'il est possible d'obtenir des accélérations parallèles super linéaires. C'est-à-dire qu'il est par exemple possible d'utiliser 4 processeurs, et de diviser le temps de calcul par un nombre supérieur à 4. Pour terminer ce chapitre, nous ferons une description de sept solveurs parallèles programmés dans un modèle à mémoire distribuée, ainsi que des sept solveurs programmés en mémoire partagée à notre connaissance aujourd'hui. Ces solveurs seront repris de manière synthétique sous forme de tableau.

Après avoir abordé tous les concepts utiles dans les chapitres précédents, nous présenterons dans le chapitre 4 les grands principes directeurs que nous avons choisis de développer pour produire un algorithme efficace à la résolution du problème SAT sur machine multi-cœurs. Nous avons appelé ce solveur SAT parallèle « MTSS » (pour *Multi-Threaded* SAT *Solver*). En particulier, nous avons choisi de résoudre les formules générées aléatoirement. Nous avons pris le parti de créer un algorithme extrêmement collaboratif afin de profiter de l'opportunité d'avoir un accès rapide à une mémoire partagée par tous les processeurs. L'approche que nous proposons est basée sur un processus riche sensiblement identique à un solveur séquentiel classique mais aidé par une multitude de processus pauvres. Ces processus pauvres sont un apport théorique nouveau car ils peuvent revêtir n'importe quel rôle pendant le processus de résolution. Le niveau de parallélisme est très élevé car nous avons amené la quantité de travail affectée à un processeur au niveau d'un traitement local à un nœud de l'arbre de recherche. Le schéma riche / pauvre n'est pas le même principe que le classique schéma maître / esclave car le processus riche contribue à la recherche de solution et ne donne pas explicitement de travail aux pauvres. Les processus pauvres ne se comportent pas de la même manière que des esclaves car ils ne font pas que le travail donné par le riche mais peuvent s'échanger du travail, au même titre qu'ils le font avec le processus riche. Il nous a fallu introduire un concept d'équilibrage de charge nouveau puisque le chemin de

guidage dans notre approche ne suffit pas à faire travailler quelques processeurs. Nous introduisons donc le principe d'arbre de guidage qui n'est plus le partage d'un chemin, mais bel et bien de l'ensemble de l'arbre binaire de recherche en mémoire centrale. Ce nouvel objet parallèle a l'avantage de permettre le travail simultané de plusieurs entités sur des nœuds différents de l'arbre de recherche. Il est donc possible d'avoir un grain parallèle très fin, autrement dit, les entités peuvent effectuer des quantités de travail très petites. Si les processus utilisés travaillent réellement en grain fin, il est inutile de recourir à une politique explicite d'équilibrage de charge puisque les tâches affectées aux processeurs sont quasi-indivisibles et qu'ils recherchent eux-mêmes du travail en explorant l'arbre de guidage.

Le chapitre 5 explique les difficultés techniques que notre approche engendre lorsque l'on souhaite effectivement implémenter le solveur MTSS ayant les caractéristiques décrites ci-dessus. Tout d'abord, il a fallu mettre en place une gestion mémoire spécifique afin de conserver de bonnes performances globales en minimisant les défauts de mémoire cache (mémoire proche d'un cœur de calcul dans le processeur) alors que l'arbre de guidage est un objet fortement partagé. La façon dont nous avons implémenter ce nouvel objet permet aux processus riche et pauvres de se transmettre de l'information en temps constant quelque soit l'endroit dans l'arbre de guidage. Nous expliquons la manière dont nous gérons les accès à l'arbre de guidage et la manière dont les processus se synchronisent. Pour se synchroniser, les processus minimisent les temps d'attente sur les verrous dans l'arbre de guidage. Pour cela, les verrous ne sont maintenus que le temps du dépôt d'une information. Lorsque le processus riche s'apprête à travailler sur un nœud en cours de calcul par un processus pauvre, il lui donne son rôle de processus riche et devient lui-même un processus pauvre. Si deux processus pauvres souhaitent travailler sur le même nœud au même instant, le premier à obtenir le verrou pourra travailler, le suivant continuera l'exploration de l'arbre de guidage à la recherche d'un autre nœud à calculer. En sus de ces caractéristiques qui permettent à MTSS de gérer un processus riche et plusieurs processus pauvres, ainsi que le développement d'un arbre de guidage, nous présentons une extension que nous avons donnée aux processus pauvres, mais également au processus riche : la possibilité de muter en un solveur externe à MTSS. De cette manière, nous donnons à disposition de la communauté de chercheurs en satisfaisabilité un outil capable de paralléliser n'importe quel autre solveur sur un ordinateur multi-cœurs. Pour utiliser au mieux ce solveur externe — même les solveurs modernes capables d'enrichir la formule de départ par des lemmes appris pendant la résolution — nous avons implémenté au sein de MTSS un mécanisme permettant d'échanger des lemmes entre les exécutions parallèles du solveur externe. Malgré le fait que MTSS soit initialement prévu pour résoudre des formules générées aléatoirement, il est donc capable de rendre parallèle n'importe quel solveur de l'état de l'art. Cette possibilité offerte par MTSS représente un travail parallèle sur des sous-arbres différents, donc un parallélisme à gros grain. Il sera également abordé la possibilité d'utiliser n'importe quelle heuristique de choix de variable de décision par l'intermédiaire d'un solveur externe capable de ne renvoyer que la racine de l'arbre de recherche qu'il s'apprête à développer.

Le chapitre 6 montre la pertinence de chaque choix que nous avons fait tout au long

du développement de MTSS. Ainsi, nous fournissons un ensemble de résultats expérimentaux montrant l'apport de la gestion mémoire spécifique, l'excellente efficacité du solveur MTSS (jusqu'à 80% sur 8 cœurs de calcul), et les bonnes performances de l'outil de parallélisation de solveurs. Nous avons effectué des tests avec des solveurs pour formules générées aléatoirement et avons constaté de très bonnes performances avec environ 60% d'efficacité sur 8 cœurs de calcul, et des résultats robustes, fiables, sans grands écarts entre différentes exécutions. Pour la parallélisation de solveurs industriels, nous avons remarqué des performances en dents de scie, mais le plus intéressant est que nous avons relevé des accélérations super linéaires pour tous types de formules, qu'elles aient une solution ou non. Nous donnons des explications et des voies d'améliorations pour ces performances moins stables que dans les précédents tests.

Le chapitre de conclusion dressera le bilan des travaux réalisés pour le solveur MTSS ainsi que des perspectives à plus ou moins long terme.

Chapitre 1

Le problème de satisfaisabilité

1.1 Introduction

Le problème de Satisfaisabilité d'une formule logique ou « problème SAT », est un problème combinatoire de décision au formalisme simple, mais possédant néanmoins un pouvoir expressif intéressant. Quelques notions de logique Booléenne sont nécessaires pour aborder le sujet. Ce chapitre débute donc par une présentation des termes utilisés pour décrire une formule Booléenne, puis par l'explication du processus de résolution de SAT et par la définition des termes utilisés dans un arbre binaire. Nous présentons des méthodes dites incomplètes qui tentent de trouver rapidement une solution à un problème SAT mais sans certitude de la trouver même si elle existe. Nous expliquons également les méthodes complètes, moins rapides en général à donner une solution, mais qui ont la capacité de toujours donner une réponse quant à la satisfaisabilité d'une formule. Nous verrons que les méthodes complètes basées sur l'algorithme DLL sont très efficaces pour résoudre des formules SAT générées aléatoirement. Ils sont à différencier des solveurs complets basés sur l'algorithme CDCL qui sont destinés à résoudre les formules décrivant des problèmes issus du monde réel, les formules industrielles. Nous présenterons les différentes caractéristiques de ces deux familles de solveurs. À savoir l'heuristique de choix de variable et les traitements locaux pour les solveurs type DLL, et l'apprentissage accompagné d'une politique de redémarrage pour les solveurs type CDCL. Concernant les formules générées aléatoirement, nous aborderons le phénomène de seuil qui permet de générer des problèmes difficiles à résoudre. Nous terminerons ce chapitre par quelques mots sur les extensions et problèmes apparentés au problème SAT.

1.2 Préliminaires

Définition 1.1 *Une **proposition** est une affirmation, elle peut-être vraie ou fausse.*

Exemple 1 *La phrase suivante : « la mer est bleue turquoise » peut-être vraie ou fausse en fonction de l'endroit où se trouve l'orateur.*

Définition 1.2 *Une **variable Booléenne**, ou variable propositionnelle peut prendre la valeur* VRAI *ou la valeur* FAUX*, elle représente une proposition logique.*

Exemple 2 *Soit x une variable Booléenne prenant soit la valeur* VRAI *soit la valeur* FAUX*. Soient A et B deux actions différentes. Selon la valeur de x, il ne sera pas fait la même action par cette suite d'instructions :*
Si x, alors faire A
Sinon, faire B

Définition 1.3 *L'**affectation** associe une valeur de vérité dans l'ensemble {* VRAI, FAUX *} à une variable Booléenne. Cette action est aussi appelée un assignement. Dans les algorithmes de ce document, nous représentons l'affectation par $\leftarrow$.*

Exemple 3 *Soient a et b deux variables Booléennes et c la conjonction $a \wedge b$. En affectant la valeur* VRAI *à a ($a \leftarrow$* VRAI*) ainsi qu'à b ($b \leftarrow$* VRAI*), c vaut* VRAI*.*

Définition 1.4 *Un **littéral** est l'expression d'une variable sous sa forme positive ou sous sa forme négative. Soit b une variable, le littéral b associé aura la même valeur de vérité que celle affectée à la variable b alors que le littéral $\bar{b}$ aura la valeur opposée.*

Remarque 1.1 *Le littéral b (respectivement $\bar{b}$) est **satisfait** lorsque la variable b est affectée à* VRAI *(resp.* FAUX*); le littéral opposé est **contredit**.*

Exemple 4 *Soit b une variable affectée à* FAUX*, le littéral $\bar{b}$ est satisfait et le littéral b est contredit.*

L'algèbre Booléenne (du nom du mathématicien britannique Boole) est la branche des mathématiques qui s'intéresse aux variables Booléennes. Il existe trois opérateurs primaires spécifiques à cet algèbre : le **NON**, le **ET**, et le **OU**.

Définition 1.5 *L'opérateur unaire **NON** inverse la valeur de vérité d'une proposition. Il se note $\bar{a}$ lorsqu'il est appliqué à la proposition a ou se note $\neg a$.*

La table de vérité associée à cet opérateur selon les valeurs possibles de la variable a est la suivante :

a	$\bar{a}$
VRAI	FAUX
FAUX	VRAI

Exemple 5 *Soit la proposition a suivante : « il fait beau », $\bar{a}$ signifie « il ne fait pas beau ».*

Définition 1.6 *L'opérateur **ET** est une conjonction de propositions. Il est défini sur deux opérandes (opérateur binaire). Une **conjonction** vaut* VRAI *ssi toutes les propositions évaluées valent* VRAI*; dans le cas contraire, la conjonction vaut* FAUX*. Nous le noterons $\wedge$.*

Voici la table de vérité associée à cet opérateur selon les valeurs possibles des variables a et b :

a	b	$a \wedge b$
VRAI	VRAI	VRAI
VRAI	FAUX	FAUX
FAUX	VRAI	FAUX
FAUX	FAUX	FAUX

Exemple 6 *La proposition suivante est vraie : « la Terre est (presque) ronde $\wedge$ la Terre tourne autour du Soleil ».*

Définition 1.7 *L'opérateur binaire **OU** est une disjonction de propositions. Une **disjonction** vaut* FAUX *ssi toutes les propositions évaluées valent* FAUX *; dans le cas contraire, la disjonction vaut* VRAI*. Il se note $\vee$.*

La table de vérité associée à cet opérateur selon les valeurs possibles des variables a et b est :

a	b	$a \vee b$
VRAI	VRAI	VRAI
VRAI	FAUX	VRAI
FAUX	VRAI	VRAI
FAUX	FAUX	FAUX

Exemple 7 *La proposition suivante est fausse : « la Terre est plate $\vee$ le Soleil tourne autour de la Terre ».*

Définition 1.8 *Une **clause** est une disjonction de littéraux. Une clause est satisfaite ssi au moins un de ses littéraux est satisfait.*

Exemple 8 *La clause $x \vee y \vee \bar{z}$ est satisfaite si au moins x ou y est affectée à* VRAI *ou z à* FAUX*.*

Définition 1.9 *Une **formule sous forme normale conjonctive** (formule* CNF *pour « conjonctive normal form ») est une conjonction de clauses. La formule est satisfaite ssi toutes les clauses le sont.*

Par convention, on appellera $\mathcal{V}$ l'ensemble des variables de la formule CNF et $\mathcal{C}$ l'ensemble de ses clauses.

Exemple 9 *La formule propositionnelle suivante est une formule* CNF *:* $(v_1 \vee v_2) \wedge (v_3 \vee \bar{v_1}) \wedge (\bar{v_4} \vee v_3 \vee v_2)$

Définition 1.10 *Une **interprétation** est le fait d'affecter une valeur de vérité à chaque variable $v \in \mathcal{V}$. Une **interprétation partielle** est une affectation sur un sous-ensemble de variables de $\mathcal{V}$.*

Remarque 1.2 *Soit $\mathcal{V}$ l'ensemble des variables d'une formule* CNF. *Posons $\mid \mathcal{V} \mid = n$. Puisque chaque variable peut être affectée par* VRAI *ou* FAUX, *alors le nombre d'interprétations différentes vaut 2^n.*

Définition 1.11 *Une* ***solution*** *est une interprétation de $\mathcal{V}$ sur l'ensemble { VRAI, FAUX } satisfaisant toutes les clauses de $\mathcal{C}$.*

Définition 1.12 *Si une formule* CNF *$\mathcal{F}$ admet une solution sur $\mathcal{V}$, elle est dite* ***satisfaisable*** *(ou* SAT*), sinon elle est* ***insatisfaisable*** *(ou* UNSAT, *de l'anglais « unsatisfiable »).*

Exemple 10 *Soit la formule* CNF *$\mathcal{F}$ suivante : $(\bar{v_1} \vee v_2) \wedge (v_3 \vee \bar{v_2})$. Si $v_1 \leftarrow$ FAUX et $v_3 \leftarrow$ VRAI, la formule a la valeur de vérité* VRAI, *une solution est donc trouvée et $\mathcal{F}$ est satisfaisable.*

Exemple 11 *Soit la formule* CNF *$\mathcal{F}$ suivante : $(\bar{v_1} \vee v_2) \wedge (v_1 \vee \bar{v_2}) \wedge (v_1) \wedge (\bar{v_2})$. Aucune interprétation des variables v_1 et v_2 ne permet de satisfaire l'ensemble des clauses. $\mathcal{F}$ est insatisfaisable.*

Définition 1.13 *Soit $\mathcal{F}$ une formule* CNF. *$\mathcal{V}$ est l'ensemble de ses variables et $\mathcal{C}$ l'ensemble de ses clauses. Lorsqu'une variable $v \in \mathcal{V}$ est affectée à* VRAI *(resp.* FAUX*), $\mathcal{F}$ est* ***simplifiée*** *: $\mathcal{F}\backslash v$ (resp. $\mathcal{F}\backslash \bar{v}$). $\mathcal{C}$ peut être séparé en 3 sous-ensembles distincts : $\mathcal{C}_{v+}$ sont les clauses contenant le littéral satisfait, $\mathcal{C}_{v-}$ sont les clauses contenant le littéral contredit et $\mathcal{C}_{v\notin}$ sont les clauses ne contenant pas les littéraux de v. Les clauses de $\mathcal{C}_{v\notin}$ sont ignorées par la simplification, les clauses de $\mathcal{C}_{v+}$ sont satisfaites (elles ne représentent plus de contrainte à satisfaire) et les clauses de $\mathcal{C}_{v-}$ sont réduites du littéral contredit.*

Remarque 1.3 *Les ensembles $\mathcal{C}_{v+}$ et $\mathcal{C}_{v-}$ ne sont pas forcément distincts mais si l'intersection de ces ensembles n'est pas vide, alors les clauses contenues dans $\mathcal{C}_{v+} \cap \mathcal{C}_{v-}$ sont des tautologies car $\forall v \in \mathcal{V}$, $v \vee \bar{v}$ est toujours* VRAI.

Remarque 1.4 *Les formules* CNF *considérées dans ce manuscrit sont exemptes de clauses tautologiques.*

Exemple 12 *Soit $\mathcal{F}$ la formule suivante :*

$$c_1 = v_1 \vee \bar{v_2} \vee v_4$$

$$c_2 = \bar{v_1} \vee v_3$$

Posons $v_1 \leftarrow$ VRAI, $\mathcal{F}\backslash v_1$ est :

$$c_1 = \text{VRAI} \vee \bar{v_2} \vee v_4$$

$$c_2 = \text{FAUX} \vee v_3$$

Par conséquent :

$$c_1 = \text{VRAI}$$

$$c_2 = v_3$$

Définition 1.14 *Une clause est dite* ***unitaire*** *ssi elle ne contient qu'un seul littéral.*

Remarque 1.5 *Il est nécessaire de satisfaire toutes les clauses unitaires d'une formule* CNF $\mathcal{F}$ *pour avoir une chance de trouver une solution. Soient* $\mathcal{C}_u$ *l'ensemble des clauses* $c \in \mathcal{C}$ *unitaires et* $\mathcal{V}_u$ *les variables des littéraux qui apparaissent dans* $\mathcal{C}_u$*. La simplification de* $\mathcal{F}$ *par les variables de* $\mathcal{V}_u$ *est un processus itératif de complexité polynômiale en nombre de clauses appelé « propagation unitaire ».*

Définition 1.15 *Soient* $\mathcal{F}$ *une formule* CNF*,* $\mathcal{V}$ *l'ensemble de ses variables et* $\mathcal{C}$ *l'ensemble de ses clauses. Posons* v *une variable,* $\mathcal{C}_v$ *l'ensemble des clauses contenant le littéral* v *et* $\mathcal{C}_{\bar{v}}$ *l'ensemble des clauses contenant le littéral* $\bar{v}$*. Si* $\mathcal{C}_v$ *ou* $\mathcal{C}_{\bar{v}}$ *est vide, alors* v *est dite* ***monotone****.*

Remarque 1.6 *Soit* v *une variable monotone dans une formule* CNF $\mathcal{F}$*.* v *peut être affectée à la valeur satisfaisant le littéral contenu dans* $\mathcal{F}$ *sans contredire* $\mathcal{F}$*, tout en diminuant le nombre de clauses de* $\mathcal{F}$ *(le problème est donc plus facile à résoudre car moins contraint).*

Définition 1.16 *Le* ***problème SAT*** *est un problème de décision consistant à dire si une formule* CNF *est satisfaisable ou non.*

Définition 1.17 *Une méthode de résolution* ***complète*** *est capable de conclure sur la nature logique d'une formule* CNF*, qu'elle soit satisfaisable ou non.*

Définition 1.18 *Une méthode de résolution* ***incomplète*** *ne peut donner une réponse garantie que pour la satisfaisabilité ou seulement pour l'insatisfaisabilité. Si la méthode ne peut trouver de réponse, alors elle n'en donne aucune.*

Exemple 13 *Un algorithme incomplet pour la satisfaisabilité est capable de dire qu'une formule est satisfaisable, dans ce cas la réponse est garantie, mais ne pourra pas conclure sur la non satisfaisabilité d'une formule. Dans le cas où l'algorithme ne peut pas répondre, alors la formule admet peut-être une solution ou est peut-être insatisfaisable.*

Remarque 1.7 *Il est possible de créer des méthodes de résolution incomplètes pour l'insatisfaisabilité, mais dans ce document, sans précision, le terme d' « incomplet » sera employé pour l'incomplétude de satisfaisabilité car c'est ce type de méthode qui connaît le plus grand nombre d'implémentations.*

1.3 Méthodes incomplètes pour la recherche de solution

Un algorithme reposant sur une méthode de résolution incomplète pour la satisfaisabilité n'explorera pas l'espace de recherche de manière extensible et systématique. C'est la raison pour laquelle un tel solveur ne peut prouver la non satisfaisabilité d'une formule, il ne sera jamais certain d'avoir parcouru les 2^n interprétations possibles si la formule contient n variables. Toutefois, la durée d'un tel algorithme est bornée et

donc raisonnable pour l'être humain. Il est alors envisageable d'utiliser des méthodes incomplètes pour résoudre des problèmes de très grande taille. Différentes familles de méthodes incomplètes existent et reposent sur des méta-heuristiques : les méthodes de recherche locale (`TNM` [WL09], `UnitWalk` [HK05] `GSAT` [SLM92], `Walksat` [SKC96], `Novelty` [LH05], `TSAT` [MSG97], `HSAT` [GW93], `G2WSAT` [LWZ07], `SAPS` [HHHLB06]), les algorithmes génétiques (`GASAT` [LSH06]), les méthodes de recuit simulé ([Spe93]), les colonies de fourmis ([SB06]), etc. Il existe aussi un solveur portfolio spécialisé dans les méthodes incomplètes (`UbcSat` [TH04]). Certains solveurs incomplets dédiés au problème SAT ne reposent pas sur une méta-heuristique, tel `SP` [BMZ05]. Voici quelques détails concernant les solveurs `Walksat` (un des plus connus, reconnus et utilisés) et `Survey Propagation` (pour son originalité).

- `Walksat` : l'algorithme est donné par l'algorithme 1. En voici une brève description. Le calcul débute par une interprétation aléatoire des variables. La recherche de solution consiste à augmenter le nombre de clauses satisfaites en inversant la valeur de vérité préalablement affectée à certaines variables. Le processus de changement de valeur d'une variable est répété un nombre de fois déterminé à l'avance ou jusqu'à l'obtention d'une solution. Si le nombre d'essai est écoulé, alors le solveur ne peut prouver qu'aucune solution n'existe. C'est en cela que ce solveur est incomplet. La variable choisie pour l'inversion est celle qui, une fois inversée, satisfait le plus grand nombre de clauses parmi les variables contenues dans une des clauses contredites (sélectionnée aléatoirement). Un tel mécanisme contient cependant un défaut : il peut plonger vers un minimum local (en nombre de clauses contredites). C'est-à-dire que l'inversion d'une seule variable ne peut que dégrader la qualité de l'interprétation actuelle. La réponse à cet inconvénient vient encore du hasard, selon une certaine probabilité une seconde variable, en plus de celle choisie, sera également inversée.
- `SP` : Ce solveur crée un graphe bipartite de factorisation représentant la formule. Les nœuds symbolisent les variables et les clauses. Les arêtes représentent la présence d'une variable dans une clause. La phase de calcul revient à transmettre des messages à partir des clauses vers les variables et inversement. Les clauses indiquent aux variables la valeur qu'elles doivent prendre afin de les satisfaire. En fonction de la valeur choisie pour chaque variable, elles avertissent les clauses qu'elles ne satisferont pas. Lorsque les messages sont les mêmes pendant deux étapes successives, le système est dit stabilisé, et une solution peut être générée sur les variables fixées. Pour illustrer ce procédé, la figure 1.1 montre un graphe de factorisation stabilisé pour la formule CNF suivante :

  $$c_1 = x \vee y$$

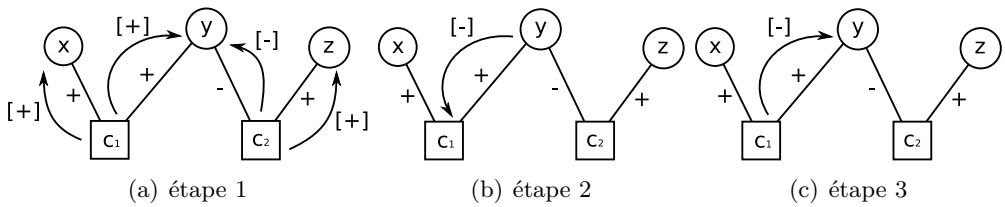
  $$c_2 = \bar{y} \vee z$$

  Les arêtes sont les lignes droites, les étiquettes '+' et '-' représentent respectivement la présence des littéraux positifs et négatifs. Les messages échangés, ainsi que leur contenu sont représentés par les flèches. Le contenu de chaque message est donné entre crochets. Ici, la solution trouvée est l'affectation de x à VRAI, y à FAUX et z à VRAI.

Algorithme 1 Procédure WALKSAT

Entrée : une formule CNF $\mathcal{F}$

```
WALKSAT(F)
pour i ← 1 à MAX_ESSAIS faire
  V_aff ← Interprétation aléatoire sur V
  pour chaque l ∈ V_aff faire
    F ← F\l (simplifications)
  fin pour
  C = ensemble des clauses contredites de F
  pour j ← 1 à MAX_INVERSIONS faire
    si C = ∅ alors
      Retourner VRAI (F est satisfaisable)
    fin si
    Choisir aléatoirement une clause c ∈ C
    si ∃ S, l'ensemble des variables x ∈ c qui, une fois inversées ne contredisent
    aucune clause de F alors
      Choisir aléatoirement une variable x ∈ S
      v ← x
    sinon
      Selon une probabilité p, faire
      v ← littéral de c qui, après inversion, contredit le minimum de clauses
      Selon une probabilité 1 − p, faire
      v ← littéral de c choisi aléatoirement
    fin si
    Inverser la valeur de v
    Mettre à jour C
  fin pour
fin pour
Retourner Échec (impossible de conclure sur la nature de F)
```



(a) étape 1 (b) étape 2 (c) étape 3

FIGURE 1.1 – Graphe de factorisation, stabilisation en 3 étapes

Il est à noter qu'une version parallèle de ce solveur destinée à être exécutée sur processeur graphique (*GPU, Graphics Processing Unit*) accélère le traitement comparé à une implémentation sur processeur classique [MZ06].

1.4 Méthodes complètes

Quelque soit la formule CNF traitée par un algorithme complet, sa nature (satisfaisable ou insatisfaisable) sera déterminée. Le nombre d'étapes nécessaire pour ce type d'algorithme est proportionnel au nombre d'interprétations possibles, et est donc d'ordre exponentiel en fonction du nombre de variables de la formule. Ces méthodes concentrent la complexité du problème. Il est par conséquent impossible d'obtenir une réponse dans un temps raisonnable si le problème est de trop grande taille. Toutefois, si la formule est satisfaisable, le solveur s'arrêtera avant d'explorer la totalité des interprétations car SAT est un problème de décision, une seule solution suffit.

1.5 L'algorithme Davis-Logemann-Loveland

La grande majorité des solveurs SAT complets aujourd'hui sont basés ou dérivés de la procédure DLL (Davis, Logemann, Loveland) [DLL62]. Cette procédure est donnée par l'algorithme 2. Elle énumère implicitement la totalité des 2^n (avec n le nombre de variables booléennes) interprétations possibles. Affecter une valeur de vérité à la variable $v \in \mathcal{V}$ permet de simplifier la formule $\mathcal{F}$ (la simplification de $\mathcal{F}$ par le littéral $\bar{v}$ est notée $\mathcal{F}\backslash\bar{v}$). De plus, la formule est simplifiée par les littéraux monotones ou unitaires. La procédure DLL énumère récursivement l'espace de recherche en procédant à des affectations successives des variables. Si une clause vide est détectée après la simplification d'une sous-formule par l'affectation d'une variable, l'algorithme effectue un retour arrière (un « Backtrack ») vers la variable la plus proche pour laquelle une valeur de vérité n'a pas encore été affectée (remontée dans les appels récursifs). Ainsi, la deuxième sous-formule sera évaluée. Au terme de ce processus, soit une solution est trouvée et est donnée à l'utilisateur (formule satisfaisable), soit toutes les combinaisons ont été implicitement testées et $\mathcal{F}$ n'admet pas de solution (formule insatisfaisable).

1.6 Arbre binaire de recherche

Une exécution de l'algorithme DLL est généralement représentée par un arbre binaire de recherche. La terminologie d'un arbre de recherche est donnée par les définitions suivantes et est représentée sur la figure 1.2.

Chaque décision de l'algorithme est représentée par un nœud interne de l'arbre et est étiqueté par le nom de la variable choisie. Les branches sortantes de chaque nœud sont les simplifications de la formule, puisque les variables peuvent être affectées par VRAI et FAUX, seules deux branches sont issues de chaque nœud. Autrement dit, soient $\mathcal{F}$ une formule CNF, $\mathcal{V}$ l'ensemble de ses variables, $\forall v \in \mathcal{V}$, si v est choisie, alors une branche issue du nœud v correspond à $\mathcal{F}\backslash v$ et l'autre à $\mathcal{F}\backslash\bar{v}$.

Algorithme 2 Procédure DLL

Entrée : une formule CNF $\mathcal{F}$

```
DLL(F)
si F contient un littéral monotone l alors
  retourner DLL(F\l) (Monotonie)
sinon si F contient une clause unitaire contenant l alors
  retourner DLL(F\l ) (Propagation Unitaire)
sinon si F contient au moins une clause vide alors
  retourner FAUX (Backtrack)
sinon si F est vide alors
  retourner VRAI (Solution)
sinon
  v ← une variable non affectée de F (Choix)
  si DLL(F\v) = VRAI alors
    retourner VRAI
  sinon
    retourner DLL(F\v̄)
  fin si
fin si
```

Définition 1.19 *Une **feuille** est un nœud de l'arbre n'ayant pas de nœuds fils, aucune branche n'est issue d'un tel nœud.*

Pendant la résolution de SAT, une feuille correspond à la découverte d'un conflit ou d'une solution, il est alors nécessaire de remonter dans l'arbre (le « backtrack ») afin d'effectuer de nouveaux choix. Une feuille peut aussi correspondre à la découverte d'une solution, dans ce cas, l'algorithme s'arrête.

Définition 1.20 *La **racine** de l'arbre est le nœud n'ayant pas de nœud père, aucune branche ne mène à ce nœud.*

La racine correspond à la première variable choisie par l'algorithme.

Définition 1.21 *Un **chemin** est une suite de nœuds depuis la racine jusqu'à un nœud n de l'arbre.*

Il représente la suite des affectations depuis le début de l'algorithme jusqu'à la décision correspondant au nœud n.

Définition 1.22 *La **profondeur** d'un nœud n est le nombre de décisions et de divisions effectuées pour amener jusqu'au nœud n.*

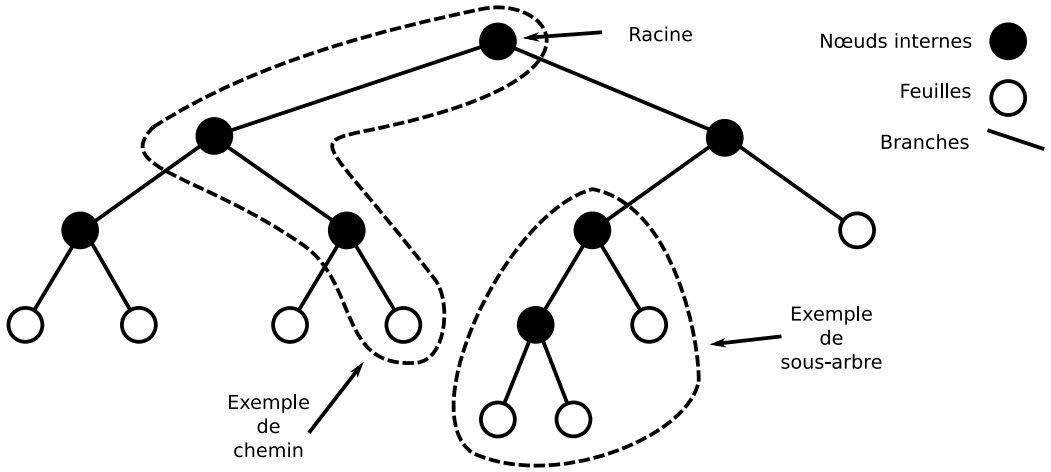


FIGURE 1.2 – Termes utilisés pour désigner les éléments d'un arbre

Exemple 14 *Soit $\mathcal{F}$ la formule suivante.*

$$c_1 = a \vee \bar{b} \vee c$$

$$c_2 = \bar{a} \vee b \vee c$$

$$c_3 = b \vee \bar{c}$$

$$c_4 = \bar{b} \vee \bar{c}$$

L'arbre de recherche correspondant à une exécution de l'algorithme DLL *qui choisit les variables dans l'ordre lexicographique est donné par la figure 1.3 (la propagation unitaire et la recherche de littéraux monotones sont prises en compte).*

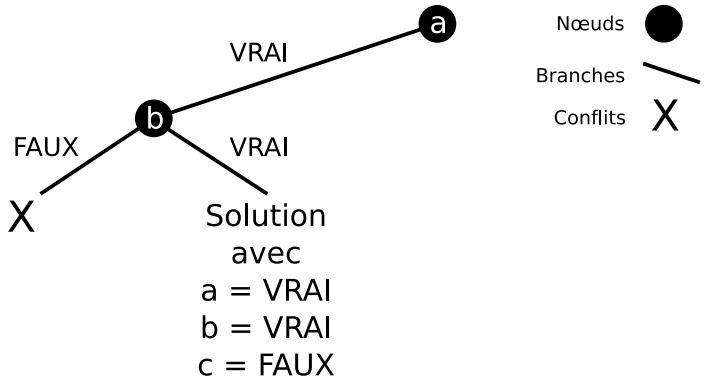


FIGURE 1.3 – Représentation arborescente de l'exécution de l'algorithme DLL

1.7 Modifications de DLL

Le grand principe de DLL est de faire des affectations successives, si le temps de calcul reste exponentiel, sans ajout de clauses l'occupation mémoire est polynômiale, contrairement à certaines autres méthodes telles que [DP60]. Beaucoup de raffinements de DLL existent, les différences se situent principalement au niveau du choix de la variable de décision ainsi que des traitements locaux appliqués à la formule à chaque nœud de l'arbre.

1.7.1 Heuristiques de choix de variables

Le but d'une heuristique efficace est de diminuer la hauteur de l'arbre de recherche, le nombre de choix est diminué et l'explosion combinatoire l'est d'autant plus. La difficulté pour réaliser une heuristique efficace est de savoir doser le temps nécessaire au choix de la variable comparé au nombre de nœuds économisés. En effet, une heuristique trop longue en temps, même si elle réduit énormément la taille de l'arbre sera peut-être moins performante qu'une heuristique beaucoup moins coûteuse en temps mais qui générera plus de nœuds (vrai en pratique mais pas en théorie, puisque dans l'absolu, il est toujours préférable de réduire le nombre de nœuds). Voici plus en détails quelques heuristiques de la littérature : MOMS, Jeroslow & Wang, UP et BSH.

1.7.1.1 MOMS

Maximum number of Occurrences in clauses of Minimum Size [DLL62, Gol79] : cette heuristique simple consiste à choisir la variable présentant le plus grand nombre d'occurrences parmi les clauses les plus petites.

1.7.1.2 Jeroslow & Wang

L'heuristique de Jeroslow & Wang [JW90] se propose d'équilibrer les tailles des sous-arbres issus d'un branchement de variable. Pour cela, cette heuristique est basée sur les occurrences des deux littéraux de chaque variable. Soit x une variable, x et $\bar{x}$ les littéraux associés. le score d'une variable est calculé de cette manière :

$\text{score}(x) = Cst \times f(x) \times f(\bar{x}) + f(x) + f(\bar{x}) + 1$

Cst est une constante et la fonction f est une fonction de représentativité du littéral dans la fonction.

Exemple 15 *Soient a et b deux variables contenues dans une formule* CNF $\mathcal{F}$. *Voici les nombres d'occurrences de chacun de leurs littéraux en fonction des tailles des clauses dans lesquelles ils apparaissent (*Occ_n *signifie « nombre d'occurrences dans les clauses de taille n », *Occ_1 *inutile à cause de la propagation unitaire) :*

Variables	Occ_2	Occ_3	...
a	*2*	*28*	...
$\bar{a}$	*7*	*14*	...
b	*4*	*17*	...
$\bar{b}$	*4*	*23*	...

Prenons le coefficient 10 pour Cst et l'heuristique MOMS pour fonction de représentativité f, on a donc :

$$score\ de\ a = 10 \times 2 \times 7 + 2 + 7 + 1 = 150\ et$$
$$score\ de\ b = 10 \times 4 \times 4 + 4 + 4 + 1 = 169$$

Dans ce cas, la variable b sera choisie.

1.7.1.3 UP

Unit Propagation [LA97] : cette heuristique fait le choix d'affecter les variables réduisant le plus grand nombre de clauses après leur affectation, en comptant les clauses réduites par les propagations unitaires. Afin d'équilibrer l'arbre (sous-arbres gauche et droit contenant un nombre de nœuds équivalents), la différence entre les « scores » des deux littéraux d'une variable doit être minimisée. Cette heuristique est implémentée dans les solveurs de la famille `satz` [LA97, Li00, JLU01].

1.7.1.4 BSH

Backbone Search Heuristic [DD06] : cette heuristique est profondément récursive et demande beaucoup de ressources en haut de l'arbre (afin de raffiner son choix) pour finalement en bas de l'arbre limiter au maximum les appels récursifs. La raison en est que plus le choix est fait haut dans l'arbre, plus il impacte la profondeur moyenne de celui-ci. L'objectif de BSH est de générer le plus rapidement possible des conflits. Le score à l'itération n d'un littéral est basé sur le score $n-1$ de chaque littéral contenu dans les clauses réduites par une affectation et la propagation unitaire associée. Au niveau de récursivité 0, le score d'un littéral est son nombre d'occurrences dans la formule. Cette heuristique est implémentée dans le solveur `kcnfs` [DD06].

1.7.2 Traitements locaux

La recherche de variables monotones dans l'algorithme DLL peut être considérée comme une première amélioration locale. En effet, elle n'est pas nécessaire à la résolution, mais elle permet d'en accélérer le traitement en réduisant le nombre de variables que l'heuristique de décision peut choisir.

1.7.2.1 *Look-Ahead*

Le *look-ahead* consiste à simplifier la formule par tout ou partie des variables non encore affectées et permet de rapidement connaître la valeur logique de la formule pour

chaque simplification. En d'autres termes, c'est une exploration de l'arbre de recherche en largeur, localement et temporairement. Si une clause vide est détectée pendant la simplification d'une formule $\mathcal{F}$ par le littéral l, alors le littéral $\bar{l}$ est propagé comme si il l'avait été par la dernière propagation unitaire (le *backtrack* sur cette variable ne sera pas effectué, donc un sous-arbre en moins à explorer). Le nombre de variables éligibles pour la prochaine variable de branchement se voit donc réduit et permet de diminuer le nombre de nœuds dans l'arbre. Il est possible d'intégrer le calcul d'une heuristique pendant l'étape du *look-ahead*, voire même d'y intégrer d'autres amélioration supplémentaires. C'est le cas, entre autres du picking, pickdepth et du pickback qui sont toutes les trois des méthodes intégrées au sein du *look ahead* développé par le solveur `kcnfs` [DD06].

1.7.2.2 *Picking*

Soit x un littéral toujours propagé, quelque soit la valeur donnée à la variable y lors du look-ahead de y. x peut être fixée sans modifier la valeur logique de la formule, cette méthode est aussi connue sous le nom de *necessary assignement* [SS98].

1.7.2.3 *Pickdepth*

Le *pickdepth* est une extension du *look-ahead*, il consiste en un développement rapide sur quelques variables contenues dans des clauses de longueur 2 afin de tester si un sous-arbre ne serait pas trivialement insatisfaisable. Cela permet de fixer un ensemble de variables sachant que certains choix conduisent à un conflit.

1.7.2.4 *Pickback*

Soit x une variable pour laquelle un test de look-ahead est effectué. Si la sous-formule générée n'est pas insatisfaisable, elle l'aurait peut-être été si x avait été affectée à la suite d'une propagation unitaire. Appelons $\mathcal{C}_x$ l'ensemble des clauses de longueur 2 satisfaites par l'affectation courante de x. $\mathcal{C}_x$ est analysé pour trouver un ensemble de variables Y_x non affectées. Pour chaque variable $y \in Y_x$, la valeur contredisant la clause de $\mathcal{C}_x$ contenant x et y est affectée à y et la sous-formule générée est analysée pour détecter un conflit. Comme dans tout look-ahead normal, si un conflit est trouvé, un ensemble de variables peut être fixé.

1.8 Formules générées aléatoirement

La procédure DLL est la base d'un grand nombre de solveurs séquentiels, les concepts de base sont conservés mais les différences sont la façon de choisir la variable à affecter ainsi que diverses améliorations locales. Les solveurs ayant conservé un comportement proche de celui de DLL sont aujourd'hui des solveurs plus naturellement dédiés à résoudre des formules générées aléatoirement. De telles formules sont en général difficiles à résoudre, même pour un petit nombre de variables. Quelques centaines de variables

suffisent pour mettre à mal les meilleurs algorithmes actuels sur les meilleurs processeurs actuels [DD03].

1.8.1 Modèle *k*-SAT aléatoire

Les formules contiennent un nombre n de variables et un nombre m de clauses. La composition des clauses est *a priori* quelconque mais il existe un modèle aléatoire appelé k-SAT aléatoire qui impose que chaque clause soit constituée d'exactement k littéraux. Les clauses sont tirées avec remise aléatoirement parmi les $2^k C_n^k$. Il est remarquable qu'il existe un algorithme de complexité polynômiale pour résoudre les formules CNF 2-SAT aléatoire. En effet, une telle formule est transformable en un problème de recherche de composantes connexes dans un graphe orienté et il existe un algorithme polynômial pour résoudre ce problème. À partir de $k \geq 3$, le problème k-SAT aléatoire $\in \mathcal{NP}$, répondre à la question de la satisfaisabilité d'une formule k-SAT requiert donc un nombre exponentiel d'étapes par rapport au nombre de variables. Le problème 3-SAT est *$\mathcal{NP}$-complet*, cela signifie que $\forall Pb \in \mathcal{NP}$, les instances du problème Pb sont transformables en temps polynômial en une formule 3-SAT. Le problème 3-SAT ayant été le premier problème à être prouvé *$\mathcal{NP}$-complet* par Cook en 1971 [Coo71], il est considéré comme le problème père des problèmes $\mathcal{NP}$. À ce titre, il est au cœur de la fameuse conjecture $\mathcal{P}? = \mathcal{NP}$. Si il existait un algorithme A pour résoudre 3-SAT en temps polynômial, alors $\mathcal{P} = \mathcal{NP}$ puisque $\forall Pb \in \mathcal{NP}$, il existe Pb' le problème Pb transformé en temps polynômial en un problème 3-SAT équivalent. Pb' serait résolvable par A en temps polynômial, on a bien $\mathcal{P} = \mathcal{NP}$. Toutefois, aujourd'hui, personne n'a prouvé que $\mathcal{P} \neq \mathcal{NP}$ mais beaucoup de chercheurs en sont convaincus. Pour plus de détails sur la complexité en général et sur la *$\mathcal{NP}$-completude* en particulier, référez-vous à l'ouvrage [Lav09].

1.8.2 Phénomène de seuil du modèle *k*-SAT aléatoire

Soient n le nombre de variables d'un problème, m le nombre de clauses associées et $c = \dfrac{m}{n}$. Il a été constaté dans la littérature [MSL92] que la probabilité pour qu'un problème k-SAT aléatoire soit satisfaisable est liée à la valeur de c. Soit p la probabilité pour qu'une formule CNF soit satisfaisable, on a :

$$p \to \begin{cases} 1 & \text{si } c \to 0 \\ 0 & \text{si } c \to \infty \end{cases}$$

Cette probabilité change brutalement en fonction de c (voir figure 1.4). Il existe un seuil pour lequel la probabilité vaut 0,5 et autour de ce seuil, le changement est très rapide. À l'image du phénomène de transition de phase en physique de la matière où un matériau change d'état subitement, une formule voit sa nature changer brusquement lorsque ce seuil est atteint, il est donc aussi appelé seuil de transition de phase. La valeur de ce seuil pour le modèle 3-SAT aléatoire est d'environ 4,25 (aucune preuve formelle n'existe à ce jour, mais seulement des études empiriques [DC91, Dub91, CR92, Goe92, MSL92, GW94a, GW94b, GW94c, GMPW95,

GW96b, GW96a, Fri99, MZK+99, Dub01, BBC+01, DD06]). Il est remarquable que la difficulté à résoudre une formule k-SAT aléatoire est liée au seuil. En effet, plus la probabilité de trouver une solution est grande, plus les algorithmes la trouve facilement, et plus la probabilité pour que la formule n'admette pas de solution est grande, plus l'algorithme a de facilités à trouver des conflits et termine rapidement l'arbre de recherche. Les problèmes aléatoires les plus difficiles à résoudre sont donc ceux générés en respectant le seuil associé à la valeur de k puisque le nombre de nœuds nécessaires à l'obtention d'une réponse explose, voir figure 1.5.

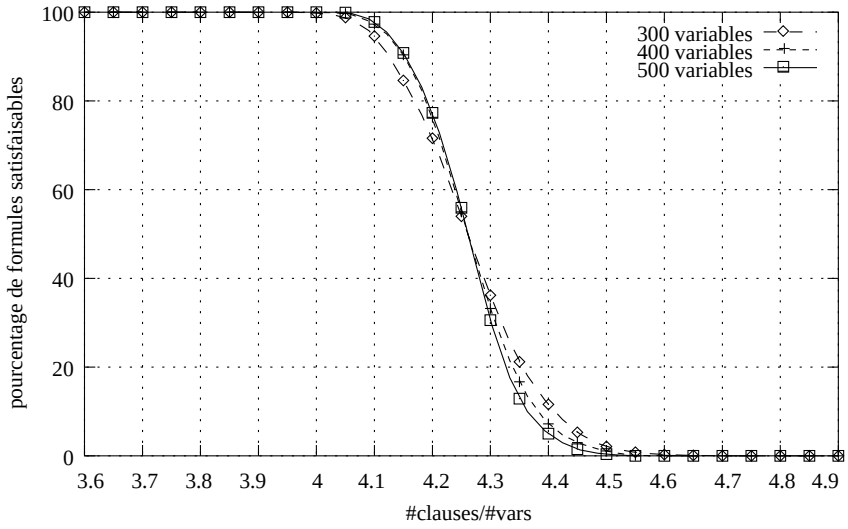


FIGURE 1.4 – Transition de phase pour le problème 3-SAT

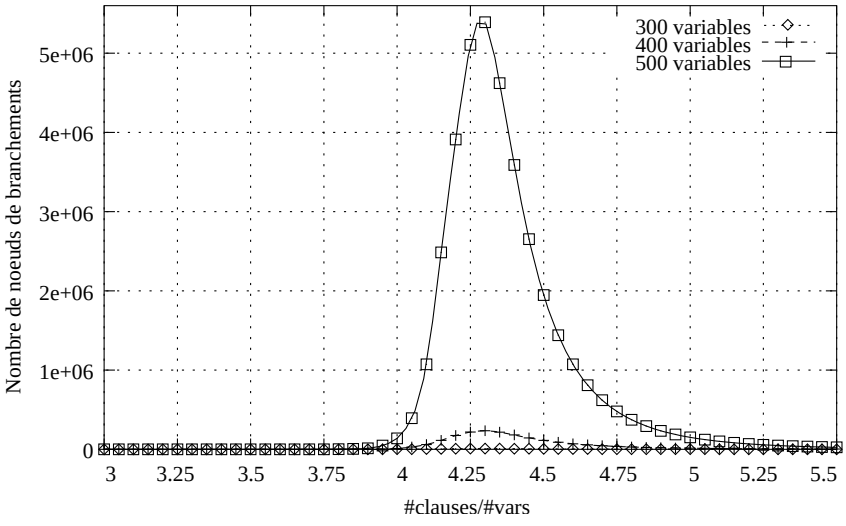


FIGURE 1.5 – Pic de difficulté pour le problème 3-SAT

1.9 Les problèmes réels

Malgré le formalisme très simple de SAT, il est utilisé pour résoudre certains problèmes industriels tels que la vérification de circuits électroniques [BCCZ99], l'intelligence artificielle (un exemple en [Lan05]), l'ordonnancement [KS98], la cryptographie [PRR$^+$07], ... Il existe des manières de générer aléatoirement des formules ressemblant à une formule issue d'un problème réel par l'exploitation de structures particulières dans les clauses [Moh09].

Les solveurs destinés à résoudre ce type de formules sont capables de résoudre des formules de plusieurs centaines de milliers de variables. Ces solveurs sont dérivés de la procédure DLL mais ont en commun un certain nombre de mécanismes qui les ont éloignés de cette procédure originelle. ils sont aussi appelés solveurs modernes et sont basés sur la procédure CDCL (*Conflict Driven Clause Learning*) donnée par l'algorithme 3 et introduite par le solveur `zchaff` [ZMMM01]. Tout d'abord, pour gérer des formules si grosses, ces solveurs utilisent des structures paresseuses pour modéliser en mémoire la formule CNF. Mais la particularité la plus importante de ce type de solveurs est la capacité d'apprendre de nouvelles clauses afin de couper plus rapidement l'arbre de recherche dans les explorations futures et ainsi gagner du temps. Les solveurs utilisent la logique contenue dans ces formules codant des problèmes réels. De plus, il s'avère que ce type de solveurs réagit très bien lorsque l'on recommence le processus de recherche de solution en ayant pris soin de sauvegarder certaines informations utiles, le processus de recherche en est amélioré. Nous allons maintenant approfondir ces notions.

1.9.1 Apprentissage

L'apprentissage, sans doute l'amélioration majeure des solveurs SAT modernes, consiste à ajouter une clause dans la base de clauses après chaque conflit rencontré. Cette technique a fait l'objet de nombreuses études [SS96, CA96, JS97, ZMMM01, MMZ$^+$01, ES03].

Un graphe d'implication est construit afin de savoir pourquoi une clause vide a été générée par le processus de résolution et les variables menant au conflit peuvent être regroupées au sein d'une clause pour extraire le sens de ce graphe d'implication. Les nœuds de ce graphe sont étiquetés par les littéraux propagés dans la formule CNF. Tout nœud n'ayant pas d'arc entrant est une variable qui a été choisie par l'heuristique de branchement (variable de décision). Il y a donc une variable de ce type par niveau de décision. Les autres variables sont affectées par propagation unitaire. Un arc orienté reliant un nœud i vers un nœud j signifie que l'affectation de i a une incidence sur l'affectation de j, la clause c qui a propagé j contient donc toutes les variables i_j telles que i_j ont été affectées auparavant mais aux valeurs contredisant les littéraux de c (voir figure 1.6).

Définition 1.23 *Un **conflit** se caractérise par la présence dans le graphe d'implication des deux littéraux d'une même variable.*

Définition 1.24 *Soit d le nœud représentant la variable de décision à la profondeur*

Algorithme 3 Procédure CDCL

Entrée : une formule CNF $\mathcal{F}$

```
CDCL(F)
V_aff ← ∅, profondeur ← 0
tantque VRAI faire
  pour chaque l ∈ V_aff faire
    F ← F\l
  fin pour
  si ∃ une clause vide alors
    si profondeur = 0 alors
      Retourner FAUX
    fin si
    c ← la clause Conflit déduite
    l ← littéral ∈ c affecté à la profondeur du conflit
    profondeur ← max{ profondeur(v), ∀v ∈ c\l }
    V_aff ← V_aff\ { ∀v / profondeur(v) ≥ profondeur}
    V_aff ← V_aff.l
    F ← F ∪ c
  sinon
    si V_aff = V (V est l'ensemble des variables de F) alors
      Retourner VRAI
    fin si
    l ← choisir un littéral d'une variable non affectée
    V_aff ← V_aff.l
    profondeur ← profondeur +1
  fin si
fin tantque
```

p. Un nœud n de profondeur p ***domine*** *un nœud m si et seulement si tous les chemins depuis d vers m passent par n.*

Définition 1.25 *Un* ***point d'implication unique*** *(UIP pour « Unique Implication Point ») est un nœud dominant le conflit.*

Remarque 1.8 *Le nœud de décision est un point d'implication unique trivial.*

Remarque 1.9 *Les UIP peuvent être ordonnés en fonction de leur distance avec le conflit. Le premier point d'implication unique (FUIP pour « First Unique Implication Point ») est l'UIP le plus proche du conflit.*

Définition 1.26 *Le* ***côté conflit*** *du graphe est un ensemble de nœuds contenant les nœuds du conflit.*

Définition 1.27 *Le* ***côté raison*** *du graphe est un ensemble de nœuds contenant les nœuds de décision.*

Définition 1.28 *Une* ***coupure*** *est une division du graphe en deux zones : le côté conflit et le côté raison. La coupure se matérialise sur tous les arcs provenant du côté raison vers le côté conflit.*

Pour éviter de retrouver les mêmes circonstances de conflit, il suffit d'ajouter à la base de clauses de la formule en cours de résolution une clause constituée des négations des littéraux à la frontière de la coupure, côté raison. En effet, c'est la conjonction de ces littéraux qui a mené au conflit, donc il ne faut pas retrouver ces affectations plus tard dans l'arbre de recherche. Le plus simple est de créer la négation de cette conjonction : la disjonction des négations. De cette manière, si la *quasi* totalité les littéraux de cette clause sont contredits, par propagation unitaire, le dernier sera satisfait. Le conflit ne sera alors pas rencontré une deuxième fois pendant la recherche de solution.

La figure 1.6 reprend ces termes et montre quelques clauses apprises à partir des coupures. Selon les stratégies mises en œuvre par les solveurs, les clauses apprises ne sont pas les mêmes. L'ensemble des clauses ayant permis de construire un tel graphe sont données dans l'exemple 16.

Exemple 16 *Clauses permettant de créer le graphe d'implication en figure 1.6 :*

$$\begin{array}{ccc} a \vee b & \bar{c} \vee \bar{d} & \bar{b} \vee d \vee e \\ d \vee \bar{e} \vee f & \bar{f} \vee g & \bar{f} \vee \bar{h} \\ \bar{f} \vee i & \bar{g} \vee h \vee \bar{k} & \bar{h} \vee i \vee k \end{array}$$

Pour construire le graphe, $\bar{a}$ a été choisi en premier, puis c.

Ce type de mécanisme permet d'éviter des conflits en forçant la propagation de certaines variables et ainsi économiser l'exploration de sous-arbres qui aurait mené à des conflits. L'inconvénient est que le nombre de clauses apprises peut devenir critique

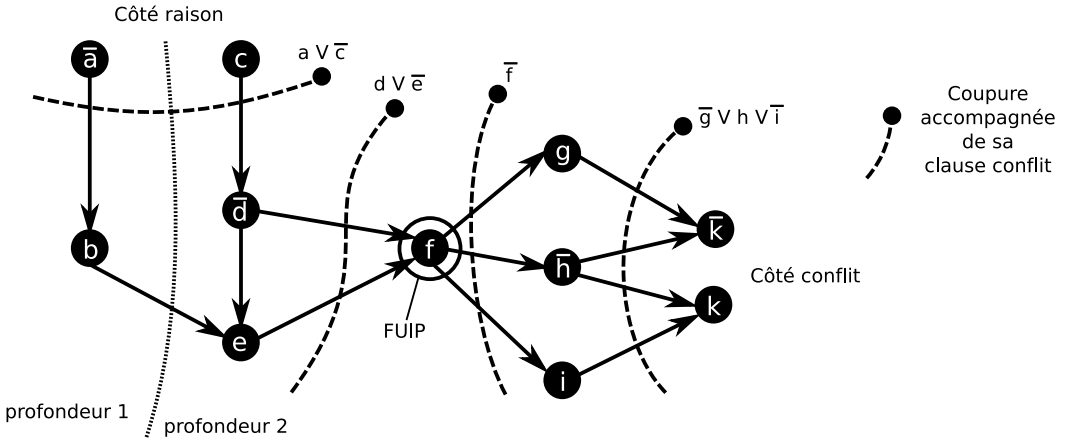


FIGURE 1.6 – Graphe d'implication et clauses apprises

et prendre beaucoup de place en mémoire. La solution à ce problème est de donner un score d'activité à ces clauses. Cette activité est augmentée à chaque fois que la clause a été utilisée et est périodiquement diminuée. Dans le cas où ce score est trop bas, la clause sera effacée lors du prochain compactage de la base.

D'autres traitements peuvent être apparentés à de l'apprentissage. Il est par exemple possible de générer de nouvelles clauses, plus courtes, mais ayant la même valeur logique sur un sous-arbre particulier. C'est ce qui s'appelle la subsomption [Dar08].

1.9.2 *Backtrack* non chronologique

Les clauses apprises servent aussi à effectuer les backtracks dans l'arbre de recherche, si bien que le retour arrière peut se faire sur plusieurs niveaux de décision (profondeurs) en un instant et remplacer la variable de décision à ce niveau de retour par la variable d'UIP choisie pour la clause apprise. Une telle clause est appelée une clause d'assertion et peut être effacée après utilisation.

1.9.3 Activité des variables

L'heuristique de branchement qui a les meilleures performances aujourd'hui sur les formules industrielles est celle se basant sur l'activité des variables dans les conflits rencontrés [ZMMM01]. Les variables ayant conduit au conflit (on les retrouve dans les graphes d'implication) voient leur activité augmentée afin qu'elle soit choisie plus facilement par l'heuristique lors de prochains choix. Selon la stratégie déployée par le solveur, l'augmentation peut être faite par une constante, mais aussi par une variable, elle même augmentée de manière exponentielle, ... afin de favoriser les variables rencontrées dans les conflits les plus récents (heuristique VSIDS [MMZ+01, MFM04]). L'idée sous-jacente à cette heuristique est que pour trouver une solution ou pour prouver la non satisfaisabilité plus rapidement, il est préférable de trouver les conflits le plus vite possible plutôt que de développer énormément de sous-arbres en vain. Les activités des

clauses sont réduites régulièrement pendant le processus de résolution à l'aide d'une division par une constante. Plus récemment, une extension à VSIDS implémentée dans le solveur glucose [AS09], propose de donner encore plus d'importance aux variables ayant entraîné le dernier conflit, issues de la dernière propagation unitaire dans les clauses dont un score particulier (nommé le LBD pour *Literals Blocks Distance*) est plus petit que celui de la clause apprise.

1.9.4 Redémarrage

Il peut sembler contre nature de vouloir redémarrer entièrement le processus de résolution, mais finalement, c'est une méthode qui fonctionne bien pour les formules industrielles. En effet, grâce aux activités des variables et des clauses, ainsi qu'aux clauses apprises par le solveur (ces informations sont conservées après un redémarrage), le solveur fera de meilleurs choix en haut de l'arbre de recherche, cela aura pour conséquence de créer un arbre de recherche compact et permettra de donner rapidement une réponse au problème. Là encore, chaque solveur a ses caractéristiques propres, certains redémarrent dès le $100^{ème}$ conflit rencontré puis augmentent cette valeur, d'autres tous les 16000 conflits [ES03, Rya04].

1.9.5 Pré traitement

Pour résoudre les formules industrielles, il est répandu d'utiliser un algorithme destiné à pré-traiter la formule CNF qui doit être résolue. Cette étape permet de gérer les cas triviaux, de réduire la formule en retirant les clauses redondantes ou ayant un sens logique plus étendu qu'une autre clause plus contraignante (la subsomption, comme par exemple dans [DDD+05]), d'ajouter des clauses intéressantes pour la résolution (en particulier des clauses binaires permettant d'augmenter le nombre de propagations unitaires), rechercher des équivalences entre les variables, d'utiliser les structures logiques entre les variables (dans `LSAT` [OSM02]) ... L'algorithme de pré-traitement le plus utilisé aujourd'hui est sans doute `SatElite` [EB05]. Le but de ce pré-traitement est de faciliter la tâche des solveurs qui seront utilisés par la suite. Dans ce cas, le temps de calcul qui fait référence est la somme du temps alloué au pré-traiteur et du temps que met le solveur à fournir une réponse.

1.9.6 Structures paresseuses

Les structures de données pour modéliser un problème SAT de très grande taille peuvent prendre énormément de place en mémoire centrale, mais le plus problématique est le nombre d'objets et leur emplacement dans ces structures à mettre à jour après propagation unitaire. En effet, généralement, assez peu de clauses sont à réduire ou à satisfaire comparé au nombre total de clauses. De plus elles ne sont pas ordonnées en mémoire par rapport aux littéraux les constituant, le travail à effectuer sur chacune d'entre elle est donc dispersé sur la totalité des structures enregistrées. Tout cela amène très peu de localité mémoire au niveau de la mémoire cache des processeurs. Les structures paresseuses permettent de mettre à jour beaucoup moins de clauses qu'avec des

structures classiques, mais l'inconvénient est que la formule est partiellement mise à jour à un instant t. Toutefois, une clause est détectée dès qu'elle devient vide. Seule la taille réelle de cette clause (si elle est non nulle) n'est pas toujours exactement connue par l'algorithme, c'est pourquoi toutes les heuristiques basées sur les tailles de clauses, comme la majorité des heuristiques utilisées dans les solveurs pour formules générées aléatoirement, ne sont plus utilisables dans un tel cadre. Ces structures de données sont donc très efficaces pour résoudre des formules industrielles mais réduisent la qualité de la connaissance que nous avons d'une formule à un instant donné [ZMMM01].

Exemple 17 *La méthode appelée « 2-watched literals » permet de ne pointer que deux littéraux par clause. Lorsqu'un de ces deux littéraux est contredit, alors une variable non affectée est cherchée pour remplacer ce littéral. Pendant cette recherche, plusieurs cas sont possibles :*

- *une variable non affectée est trouvée, elle remplace donc la variable dans la structure des 2-watched literals dont le littéral a été contredit.*
- *un littéral satisfait est découvert, la clause peut être ignorée à l'avenir car elle est satisfaite.*
- *toutes les variables sont déjà affectée et les littéraux sont tous contredits : il faut satisfaire et propager le deuxième littéral pointé par la structure 2-watched literals.*

1.10 Extensions et problèmes apparentés

Le problème SAT, en tant que premier problème prouvé $\mathcal{NP}$*-complet*, est considéré comme le père des problèmes combinatoires de décision. Il existe une multitude de déclinaisons de ce problème de décision : les problèmes de maximisation ou d'optimisation associés par exemple. Voici une liste non exhaustive d'extensions ou de problèmes apparentés au problème SAT :

1.10.1 # SAT

Il s'agit de dénombrer les solutions possibles à une formule CNF. Que la formule soit satisfaisable ou non, la totalité de l'espace de recherche sera exploré. Contenir l'arbre de recherche en mémoire est difficile pour ce type de problème.

1.10.2 Max-SAT

Ce problème consiste à trouver l'interprétation qui engendre le minimum de clauses contredites (clauses vides). Ce problème n'a de sens que pour les formules insatisfaisables car en présence d'une solution, le nombre de clauses vides est zéro. Les solveurs peuvent par exemple travailler à partir d'une borne maximum calculée par une heuristique puis explorer l'arbre de recherche. Pendant la recherche, si le nombre de clauses contredites dépasse la valeur courante de la borne maximum, alors le solveur peut *backtracker*, il sait qu'il n'améliorera pas la borne dans le sous-arbre actuel. Pour de plus amples informations sur Max-SAT, voir [Dar08, Moh09].

1.10.3 Max-SAT pondéré

Même problème que le précédent, seulement ici ce n'est pas le nombre de clauses contredites qu'il faut minimiser mais la somme des poids qui leur sont affectés. Autrement dit, cette variante permet de mettre des priorités sur les clauses.

1.10.4 QBF

QBF : formule booléenne quantifiée.

Pour le problème SAT, chaque variable est existentielle, c'est-à-dire qu'il suffit de trouver une valeur pour chaque variable qui satisfasse la formule et le problème a trouvé une solution. Pour le problème QBF, c'est différent puisqu'il peut contenir des variables universelles. Notons $\mathcal{U}$ l'ensemble de ces variables, un solveur QBF devra trouver une solution pour chaque valeur des variables de $\mathcal{U}$. Pour chaque $u \in \mathcal{U}$, il est nécessaire de résoudre deux fois le problème SAT, une fois en affectant u à VRAI et une fois en l'affectant à FAUX. SAT est donc un cas particulier de QBF où $\mathcal{U} = \emptyset$.

1.10.5 CSP

Les CSP (problème de satisfaction de contraintes) permettent une sémantique plus grande que SAT. Au lieu d'être constitué de variables binaires, ici les variables peuvent recevoir un nombre quelconque de valeurs. De plus, chaque variable a son propre ensemble de valeurs possibles. On distingue les CSP discrets à valeurs entières et les CSP à valeurs réelles. SAT est donc un sous-ensemble de CSP : c'est un CSP binaire où les seuls opérateurs sont le ET ($\wedge$) ainsi que le OU ($\vee$).

1.10.6 Physique statistique

Le problème SAT a certaines similitudes avec les problèmes de physique statistique [PIM06]. Par exemple, le phénomène de seuil est intéressant pour la communauté des physiciens et plus généralement, les aspects de classes de problèmes intéressent les physiciens et les mathématiciens.

1.11 Compétition

Vue l'importance de résoudre SAT efficacement, des compétitions existent telles que la « SAT competition [1] » ou encore la « SAT-race [2] ». Un ensemble de solveurs est testé sur un ensemble de formules appelées des *benchmarks*. Les compétitions distinguent les résultats sur les formules aléatoires, industrielles ou académiques (*handmade* ou *crafted*), que ce soit sur les problèmes satisfaisables ou insatisfaisables. Ainsi, le comportement de chaque solveur peut être apprécié à sa juste valeur et il est plus simple de détecter quelle méthode fonctionne pour quel type de formule. La dernière compétition à la date de rédaction de ce document a été la SAT competition 2009. Les gagnants sont :

1. `http://www.satcompetition.org/`
2. `http://baldur.iti.uka.de/sat-race-2008/` pour la plus récente

- catégorie formules industrielles, SAT ou UNSAT : `precosat` [Bie09], un solveur ayant beaucoup de points communs avec `Minisat` [SE08] (le solveur le plus utilisé aujourd'hui en tant que base) et au pré-traiteur `SatElite` [EB05].
- catégorie formules académiques, SAT ou UNSAT : `clasp` [GKS09], similaire encore une fois au couple `Minisat` et `SatElite`.
- catégorie formules aléatoires, SAT ou UNSAT : `SATzilla2009_R` [NDS+04, XHHLB09], un logiciel port-folio de solveurs SAT, il analyse la formule et lance le solveur qui lui semble être le plus adapté. On ne sait pas quel solveur il a exécuté, mais probablement des solveurs incomplets du type `gnovelty+` [PG07] ou `adaptg2wsat0` [LZ07] dans un premier temps, puis les solveurs complets `kcnfs` [DD03] ou `March` [HvM06] pour prouver l'insatisfaisabilité si les solveurs incomplets ne donnaient aucune réponse. Les quatre solveurs cités ci-dessus sont parmi les plus performants aujourd'hui pour ce type d formule.

1.12 Conclusion

Comme montré dans ce chapitre, SAT a beaucoup d'applications réelles ou de défis théoriques à relever. Cet axe de recherche intéresse aussi bien les industriels pour la grande variété des problèmes modélisables ainsi que les bonnes performances des solveurs, mais aussi les universitaires pour les aspects fondamentaux. Tout progrès dans les performances de résolution de SAT accélère donc de nombreux autres problèmes. De plus SAT est apparenté à beaucoup d'autres problèmes, et l'amélioration de SAT peut être une piste pour développer de nouveaux algorithmes performants pour ces problèmes (comme par exemple # SAT ou Max-SAT). SAT est un domaine de recherche particulier car la recherche expérimentale y a une part non négligeable (par exemple la découverte du phénomène de seuil). D'ailleurs, grâce aux découvertes théoriques mais aussi empiriques, l'algorithmique séquentielle pour SAT a fait de grand progrès. Cependant depuis quelques années les avancées ne sont plus toujours aussi significatives qu'auparavant. De plus, les processeurs d'aujourd'hui n'augmentent plus leur vitesse d'exécution, pour l'instant bloquée à un peu moins de 4 Ghz depuis 2005. Cela fait maintenant 4 ans que les processeurs grand public deviennent petit à petit des processeurs parallèles en intégrant plusieurs cœurs de calcul. Le parallélisme a toujours été un bon moyen de gagner en vitesse, mais les chercheurs intéressés par SAT ont souvent privilégié la programmation séquentielle. Quelques tentatives de solveurs SAT parallèles existent et fonctionnent plutôt bien, ils seront d'ailleurs développés au chapitre 3. La majorité d'entre eux est restée au stade de projet ou seulement de première version car ils ne sont pas maintenus par leurs créateurs. De plus, ces solveurs sont en général développés pour être exécutés sur des machines parallèles, ce qui limite leur utilisation sur de simples machines de bureau alors que les processeurs grand public deviennent parallèles. Il est dorénavant possible pour tous de tester la validité d'une approche parallèle plus simplement, sans recourir à de lourdes structures parallèles. Le parallélisme amène plus de flexibilité, plus de possibilités à la manière d'élaborer des algorithmes. Mais le parallélisme a ses règles, ses notions. Les grands principes du parallélisme sont

donnés au chapitre 2 qui fait office d'explication du parallélisme. Le chapitre 3 dresse une liste des concepts du parallélisme dans le cadre de la résolution de SAT. Il y sera abordé les difficultés de paralléliser SAT et les solutions apportées par la communauté.

Chapitre 2

Modèles parallèles

2.1 Introduction

Résoudre SAT rapidement est un défi et depuis que les industriels ont découvert que ce problème pouvait leur être utile, les recherches se sont accélérées. Beaucoup d'innovations ont permis d'atteindre de bonnes performances, mais ce n'est jamais suffisant, il existe des problèmes dont la taille ne leur permet pas d'être résolus aujourd'hui, même par les meilleurs solveurs contemporains. Cette affirmation sera vraie tant que la conjecture $\mathcal{P} \neq \mathcal{NP}$ n'aura pas été contredite (SAT est au cœur de cette conjecture car à l'heure actuelle SAT est dans $\mathcal{NP}$ mais pas dans $\mathcal{P}$). Pour autant, un moyen simple d'augmenter les performances d'un algorithme est de l'exécuter sur un processeur plus récent, plus véloce. En effet, d'après la loi de Moore, tous les 24 mois grâce aux avancées scientifiques et aux gravures toujours plus fines des micro-processeurs, ces derniers peuvent recevoir deux fois plus de transistors sur une même surface. Depuis le début des micro-processeurs, jusqu'aux années 2000 cette diminution de la finesse de gravure s'est donc traduite par une augmentation du nombre de transistors mais aussi par une augmentation des fréquences de fonctionnement des micro-processeurs. Cela signifie que dans un laps de temps donné, le nombre d'instructions traitées est plus grand. Depuis environ 2005, cela ne se traduit plus par une augmentation de fréquence mais par l'augmentation du nombre de cœurs. Les programmes séquentiels ne sont donc plus accélérés par une montée en fréquence. En effet, à cause de l'échauffement trop important des micro-processeurs à des fréquences approchant les 4 GHz, les entreprises ne peuvent plus fabriquer de micro-processeurs traditionnels au silicium au-delà de cette vitesse de fonctionnement. Dans le même temps, la loi de Moore est toujours vraie, et il est toujours possible d'augmenter par 2 le nombre de transistors dans un micro-processeur. La réponse à ce dilemme est l'augmentation du nombre de cœurs de calcul au sein du même micro-processeur. Ce sont les processeurs multi-cœurs (ou en anglais « multi-core »). De cette manière, les industriels annoncent toujours doubler la puissance de leurs micro-processeurs, mais les utilisateurs ne voient pas ce gain directement. En effet, la plupart des applications, et le domaine SAT ne fait pas exception, sont issues du modèle de programmation séquentiel. Ces programmes sont imaginés

pour n'être exécutés que par un seul cœur de calcul. En réalité, pour tirer parti de la puissance de ces nouveaux micro-processeurs, il faut développer dans un autre modèle de programmation : il faut penser parallèle. Ces propos sont repris et détaillés en début de ce chapitre en section 2.2.

Ensuite, ce chapitre donnera les grands principes du parallélisme en commençant par un ensemble de définitions et par le problème de l'équilibrage de charge. Suivra une explication sur les différents modèles d'exécution : une ou plusieurs instructions avec une ou plusieurs données (SISD, SIMD, MISD et MIMD). Dans le modèle le plus utilisé pour résoudre SAT, à savoir plusieurs instructions avec plusieurs données (MIMD), nous détaillons les architectures matérielles adaptées. Ces architectures sont CC-UMA, CC-NUMA et NoRMA, les mémoires sont soit partagées soit distribuées. Nous finirons avec la présentation des différents modèles de programmation PRAM et DRAM qui définissent si le programmeur fait discuter ses processeurs par l'intermédiaire d'une mémoire partagée ou par l'échange de messages. Dans le cas de la mémoire partagée, il est primordial de faire attention à la cohérence des données modifiées par plusieurs processeurs, ce point sera abordé. En résumé, ce chapitre présente les modèles de parallélisme. Le chapitre suivant fera le lien entre le parallélisme et le problème SAT.

2.2 Motivations

Les performances des solveurs SAT séquentiels n'ont cessé d'augmenter durant les dernières années, grâce à l'algorithmique bien entendu, mais également par la puissance des ordinateurs toujours plus importante. Le lecteur pourra légitimement se demander pourquoi il est nécessaire ou intéressant d'étudier les aspects parallèles du problème SAT. La réponse est simple : les ordinateurs, même les plus simples, deviennent des machines parallèles, c'est ce que nous montrons ici. Aujourd'hui il ne manque que les algorithmes parallèles pour résoudre SAT sur ce type d'ordinateurs et ce sera l'objet des chapitres suivants.

2.2.1 Loi de Moore et multi-cœurs

Dès 1965, un des co-fondateurs de la compagnie *Intel*, nommé Gordon Moore, à partir des constats de l'époque prédit que « The complexity for minimum component costs has increased at a rate of roughly a factor of two per year » en parlant des micro-processeurs. Cela signifie que le coût pour produire des micro-processeurs était divisé par 2 tous les ans. En 1975, il reprit ses affirmations pour en changer la durée de 12 mois à 24 mois. Dérivée de ces affirmations, la célèbre loi de Moore s'énonce maintenant ainsi :

Définition 2.1 Loi de Moore *: le nombre de transistors au sein d'un processeur double tous les deux ans.*

Cette loi, qui est en réalité une conjecture empirique, se révèle jusqu'à maintenant exacte, comme le montre la figure 2.1 [1]. La droite pointillée traversant la figure représente la loi de Moore et les points sont les dates de sorties de certains micro-processeurs (liste non exhaustive). En abscisse est indiquée l'année de sortie d'un micro-processeur et en ordonnée, le nombre de transistors qui le composent.

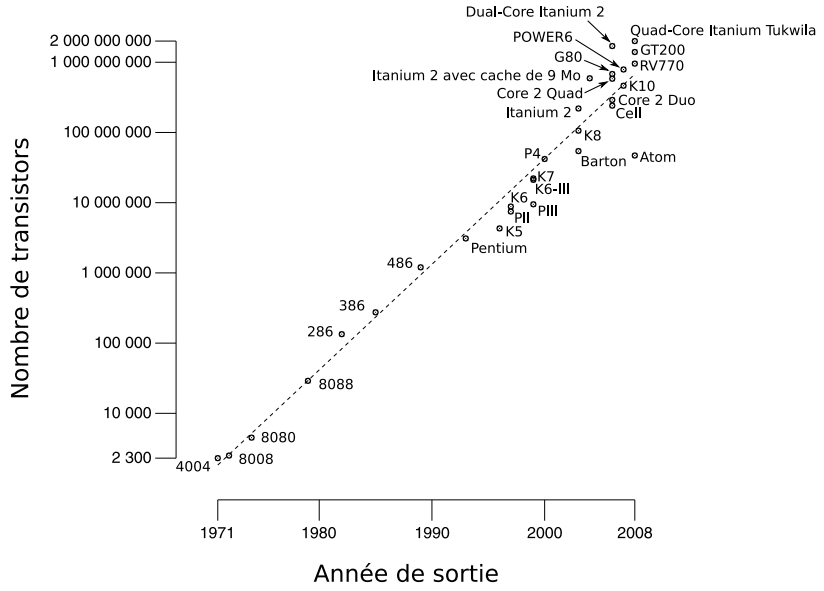


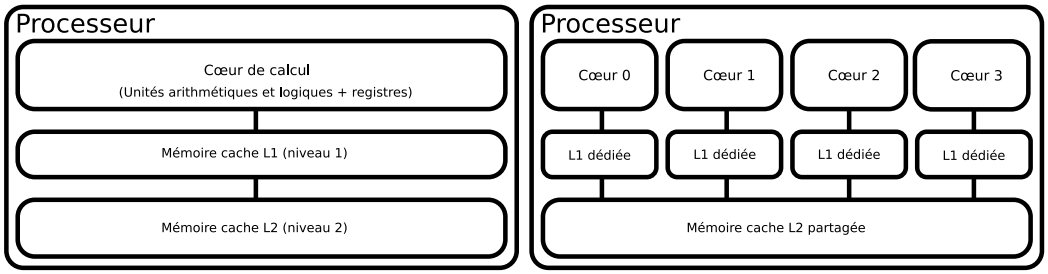
FIGURE 2.1 – Loi de Moore et réalité

Pour simplifier, on peut dire que la *puissance* des micro-processeurs double tous les 2 ans (le terme de puissance étant très général). Entre le début de la micro-informatique et peu après les années 2000, cette augmentation de puissance se traduisait par une montée en fréquence des micro-processeurs. Par conséquent, sans effort de programmation supplémentaire, il était possible d'accélérer un programme séquentiel en changeant le processeur qui l'exécute puisque pour une même durée, le nombre d'instructions traitées était plus grand. À partir d'une fréquence trop grande, les micro-processeurs dégagent tellement de chaleur qu'il devient impossible de les refroidir de façon habituelle (ventilateurs). Cette montée en fréquence a donc connu sa limite vers 2004 - 2005. À titre d'exemple un des processeurs les plus rapides (en termes de fréquence pure) de la compagnie *Intel*, le Pentium 4 672 sorti début 2005, atteignait une vitesse d'exécution de 3,8 GHz. Depuis ce processeur — à ma connaissance — *Intel* n'a pas créé de processeurs grand public plus rapide.

Depuis 2005, les progrès de gravure étaient tels qu'il était encore possible de doubler le nombre de transistors sur une puce de silicium tous les 2 ans, mais plus la fréquence. La solution apportée par les industriels a été de mettre sur un même micro-processeur plusieurs cœurs de calcul. Cette technologie se nomme les micro-processeurs multi-

1. Image modifiée (texte francisé et agrandi) issue de la page Wikipédia anglaise concernant la loi de Moore (créée par l'utilisateur Wgsimon). Sous licence *Creative Commons Attribution ShareAlike 3.0.*

cœurs (ou en anglais « multi-core »). La figure 2.2 montre les caractéristiques d'un micro-processeur à un seul cœur de calcul et d'un micro-processeur contenant 4 cœurs de calcul. Les mémoires caches sont des mémoires tampon, en général intégrées aux processeurs, offrant aux cœurs de calcul un accès très rapide aux données. Comme la figure le montre, ce qui a fondamentalement changé entre les processeurs d'avant 2005 et ceux d'aujourd'hui est le nombre de cœurs de calcul.



(a) Mono-cœur de calcul (b) Multi-cœurs de calcul

FIGURE 2.2 – Schémas de processeurs

Les nouveaux micro-processeurs sont intrinsèquement des machines parallèles. Donc oui, ils offrent globalement toujours 2 fois plus de puissance que leurs ainés de 2 ans, mais aujourd'hui, pour tirer parti de cette augmentation de puissance il est nécessaire de changer d'approche algorithmique. Il ne faut plus penser en algorithmique séquentielle mais en algorithmique parallèle. Cela dit, cet effort n'est à fournir qu'une seule fois. En effet si les algorithmes sont bien pensés et offrent une bonne capacité d'extension sur plusieurs processeurs, alors ils profiteront du même effet d'accélération de performances en fonction du nombre de cœurs que les algorithmes séquentiels par rapport à la fréquence des processeurs. Cette constatation est une première motivation pour développer un algorithme parallèle pour SAT.

2.2.2 Consommation d'énergie

Est-ce un réel intérêt des industriels, ou une façon de ne pas avouer qu'ils ne savent pas monter en fréquence ? Mais l'arrivée des processeurs multi-cœurs est aussi expliquée par une volonté de consommer moins d'énergie. Dans le contexte écologique actuel, il n'est pas superflu d'économiser de l'énergie. Les processeurs multi-cœurs permettent une économie d'énergie puisqu'à consommation égale, deux cœurs de calcul ont plus de deux fois la puissance d'un seul cœur de calcul. Ce principe est montré par la figure 2.3 [2] ainsi que par l'exemple suivant.

Exemple 18 *Voici un tableau de 2 processeurs de la firme* Intel *: le Xeon 3,6 GHz ne contient qu'un seul cœur alors que le Xeon E7450 en contient 6. Ce tableau met en avant certaines de leurs caractéristiques. En particulier, on pourra remarquer l'efficacité*

2. Le comportement décrit dans la figure a été donné par James Reinders, un directeur d'Intel, lors d'une interview pour le site ZDNet.fr.

énergétique du multi-cœurs : pour une consommation moindre, le processeur se révèle 8 fois plus puissant avec seulement 6 cœurs. La difficulté est de développer un programme qui puisse utiliser les 6 cœurs de la meilleure manière afin de réellement multiplier par 8 les performances.

Modèle	*Intel Xeon 3.60 GHz*	*Intel Xeon E7450*
Nombre de cœurs	*1*	*6*
Fréquence (GHz)	*3,6*	*2,4*
TDP (Watts)	*110*	*90*
MFLOPS	*7200*	*57600*
Rapport TDP / cœurs	*110*	*15*
Rapport MFLOPS / TDP	*65,45*	*640*

*Le nombre de **MFLOPS** (« Mega FLoating-point Operations Per Second ») représente le nombre de millions d'opérations en virgule flottante que le processeur est capable de traiter pendant une seconde. C'est un critère de puissance de calcul brut maximum. Le **TDP** (« Thermal Design Power ») ne donne pas la consommation d'un processeur, mais renseigne sur sa consommation, plus le TDP est élevé, plus le processeur est gourmand en énergie.*

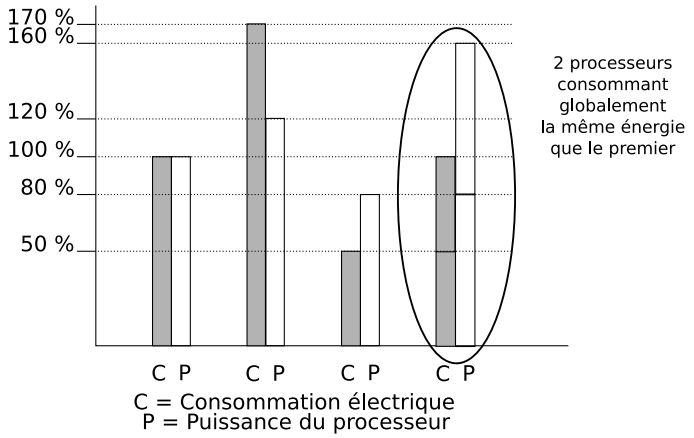


FIGURE 2.3 – Performances d'un processeur *vs* puissance électrique fournie

Les processeurs multi-cœurs offrent donc de meilleures performances pour une consommation énergétique réduite. La difficulté est d'utiliser la totalité des cœurs de calcul au mieux et nous verrons que ce n'est pas toujours facile.

2.2.3 Conclusion

Qu'elle qu'en soit la raison, le parallélisme arrive inéluctablement dans l'informatique grand public. Auparavant une spécialité pour certains domaines de recherche, comme par exemple les militaires ou les scientifiques, le parallélisme doit maintenant être pris en compte par tous les programmeurs. Il sera toujours possible de produire des algorithmes séquentiels, mais seulement si on ne souhaite pas programmer ce qui est

imaginé, ou si la rapidité d'exécution n'est pas une contrainte (pour l'enseignement par exemple). Dès l'instant que les performances ne sont pas à négliger, le parallélisme est aujourd'hui incontournable. SAT étant un domaine de recherche en Informatique très axé sur les performances doit obligatoirement prendre en compte ce point pour continuer à progresser. En outre, nous verrons au chapitre 3 que le parallélisme apporte un gain non négligeable à SAT. Nous allons maintenant présenter les différents modèles d'exécution et les modèles de programmation qui existent dans le domaine.

2.3 Préliminaires

2.3.1 Définitions

Définition 2.2 *Le **parallélisme** en Informatique donne la possibilité d'exécuter plusieurs instructions simultanément.*

Exemple 19 *L'algorithme séquentiel 4 applique la fonction f sur les n éléments d'un tableau. Cet algorithme nécessite n étapes pour se terminer.*

Algorithme 4 Algorithme séquentiel

Entrée : A : un tableau de n réels
Sortie : B : un tableau de n réels
  Exemple Séquentiel(A)
  pour $i \leftarrow 1$ à n **faire**
    B[i] $\leftarrow$ f(A[i])
  fin pour
  Retourner B

L'algorithme parallèle 5 effectue le même traitement mais utilise n processeurs. Cet algorithme calcule n valeurs par n processeurs en même temps : une seule étape est nécessaire.

Algorithme 5 Algorithme Parallèle

Entrée : A : un tableau de n réels
Sortie : B : un tableau de n réels
  Exemple Parallèle(A)
  pour chaque i (numéro du processeur) **faire**
    B[i] $\leftarrow$ f(A[i])
  fin pour
  Retourner B

Définition 2.3 *Une seule **unité de calcul** peut exécuter une seule tâche, c'est une **exécution séquentielle**. n unités de calcul peuvent exécuter n tâches distinctes simultanément : c'est une **exécution parallèle**.*

Définition 2.4 *La **granularité** du travail est la quantité de travail réalisée par chaque processeur. Soient n le nombre d'opérations à réaliser et p le nombre de processeurs, alors la granularité est* $\frac{n}{p}$.

Exemple 20 *L'algorithme 5 présente un grain très fin puisque la tâche parallélisée s'applique sur une seule case de tableau.*

Exemple 21 *Pour le problème* SAT*, un grain fin peut s'assimiler à une parallélisation des nœuds de l'arbre de recherche alors qu'un gros grain s'appliquerait à des sous-arbres.*

Définition 2.5 *Le **degré** de parallélisme correspond au nombre de tâches pouvant être calculées simultanément.*

Exemple 22 *L'algorithme 5 a un degré élevé puisque le nombre de processeurs utilisables est de la même taille que le problème. En revanche, il est inutile d'utiliser plus de processeurs que le nombre de cases dans le tableau.*

Exemple 23 *En* SAT*, le degré de parallélisme est plus important si l'algorithme permet de travailler en parallèle sur les nœuds de l'arbre de recherche plutôt que sur des sous-arbres (puisque chaque sous-arbre contient plusieurs nœuds).*

2.3.2 Éléments de comparaison de performances

Définition 2.6 *Le **temps d'exécution** est le temps écoulé entre le lancement d'un programme et l'obtention du résultat.*

Notation utilisée dans ce document :
- T_{seq} : le temps d'exécution d'un programme séquentiel.
- $T_{//}(n)$: le temps d'exécution en parallèle sur n processeurs, c'est à dire l'instant où le dernier processeur a terminé son travail (T_{seq} est un cas particulier où $n = 1$).

Définition 2.7 *L'**accélération**, notée $Acc(n)$ sur n processeurs, représente le nombre de fois que le programme a été accéléré par son exécution en parallèle et est donnée par la formule suivante :* $Acc(n) = \frac{T_{seq}}{T_{//}(n)}$.

Deux types d'accélération sont à distinguées. Soit $\mathcal{P}$ un problème quelconque, l'accélération absolue est calculée en prenant T_{seq} le temps d'exécution du meilleur algorithme séquentiel actuellement disponible pour résoudre $\mathcal{P}$. L'accélération relative est calculée en prenant $T_{seq} = T//(1)$, (le même programme). Sans précision, le terme « accélération » désignera l'accélération relative.

Définition 2.8 *L'**efficacité** d'un programme parallèle, notée $Eff(n)$ pour une exécution sur n processeurs, est définie par* $Eff(n) = \frac{Acc(n)}{n}$. *Ce nombre est l'accélération pondérée par le nombre de processeurs utilisés, il représente donc la qualité de la parallélisation en fonction du nombre de processeurs.*

Définition 2.9 *L'**iso-efficacité** d'un algorithme parallèle est la quantité de travail supplémentaire nécessaire pour garantir l'efficacité parallèle quand le nombre de processeurs augmente.*

Soit n le nombre de processeurs, et une efficacité cible e. L'iso-efficacité permet de déterminer quelle taille de problème est à traiter pour atteindre l'efficacité e sur n processeurs.

Exemple 24 *Dans le contexte de* SAT*, l'iso-efficacité correspondra à déterminer la taille des formules à résoudre si l'on souhaite maintenir par exemple une efficacité de 75% quelque soit le nombre de processeurs. Plus le nombre de processeurs sera grand, plus le problème devra contenir de variables afin de fournir assez de travail à toutes les unités de calcul.*

Définition 2.10 *L'**extensibilité** d'un algorithme parallèle est sa capacité à maintenir son efficacité quelque soit le nombre de processeurs utilisés.*

Soit f la fonction d'accélération du temps de calcul par rapport au nombre de processeurs utilisés. Si la dérivée $f'(x) > 0$ quand $x \to \infty$ alors l'algorithme est extensible puisque l'accélération augmente toujours quand le nombre de processeurs augmente. Toujours pour $x \to \infty$, si $f'(x) = 0$ alors il est inutile d'augmenter le nombre de processeurs au delà d'un certain nombre n puisque l'accélération est constante en utilisant n processeurs. Si dans un cas vraiment défavorable où la fonction dérivée est négative, cela implique que l'accélération décroît. L'algorithme n'est pas du tout extensible car utiliser un nombre trop grand de processeurs dégrade les performances. Dans ce cas, le plus utile est d'utiliser le nombre de processeurs qui maximise l'accélération. Pour mieux appréhender ces phénomènes, la figure 2.4 représente, avec une échelle linéaire, quelques courbes d'accélération types.

Définition 2.11 *Le **travail** effectué par un algorithme exécuté par n processeurs est $\sum_{i=1}^{n} Inst(i)$ avec $Inst(i)$ le nombre d'instructions traitées par le processeur i.*

Pour le problème SAT dans le contexte d'un solveur complet qui travaille sur un arbre de recherche, comme dans nos travaux, la notion de travail fera référence au nombre de nœuds développés dans l'arbre de recherche.

Définition 2.12 *Une accélération **super linéaire** est obtenue lorsque $Acc(n) > n$ avec n processeurs.*

Dans un tel cas, le parallélisme n'a pas apporté qu'un simple gain de vitesse, mais a aussi diminué le travail de l'algorithme. De telles performances sont rares et dépendent grandement du type de problème à résoudre, mais sont possibles par exemple en mettant en place un échange dynamique d'informations récoltées pendant le calcul. Dans le cas du problème SAT (et dans le cas de n'importe quel problème de décision), lorsqu'une solution est trouvée par un processeur, tous les processeurs s'arrêtent, cela peut engendrer des gains super linéaires.

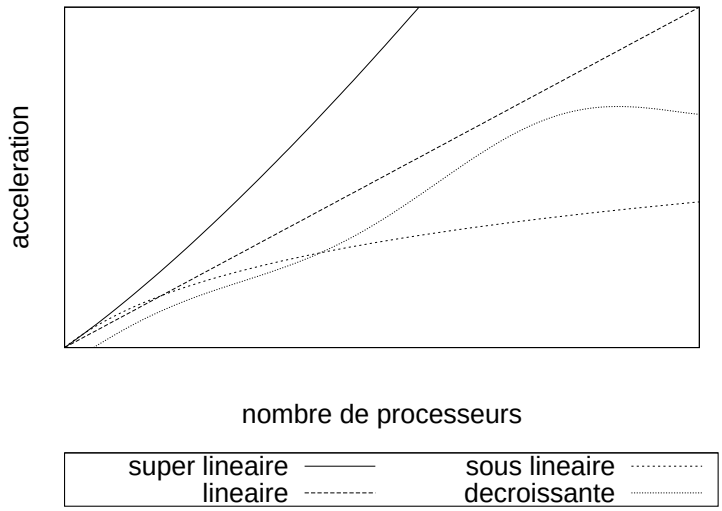


FIGURE 2.4 – Exemples d'accélérations possibles, échelle linéaire

Exemple 25 *Des performances super linéaires sont possibles pour* SAT *si le parallélisme a permis de diminuer la taille de l'arbre de recherche.*

Définition 2.13 *Gene Amdahl énonça une loi en 1967, selon laquelle, la proportion de code séquentiel d'un algorithme parallèle limite son accélération maximale. Soient s la proportion de code séquentiel (code non-parallélisable) et $Acc_{max}(n)$ la meilleure accélération sur n processeurs, cette loi est donnée par l'équation*

$$Acc_{max}(n) = \left(s + \frac{1-s}{n}\right)^{-1}.$$

Preuve: L'accélération maximum d'un programme est le quotient du temps séquentiel par le temps parallèle minimum :

$$Acc_{max}(n) = \frac{T_{seq}}{T_{min//}(n)}$$

Posons $T_{seq} = 1$ et s la proportion de code séquentiel, $T_{min//}(n)$ étant le meilleur temps parallèle, il correspond à T_{seq}/n. Toutefois, s étant la proportion de code séquentiel, lorsque ce code est exécuté par un seul processeur, malheureusement les $n-1$ processeurs travaillent quand même à ce moment et ne divisent pas s. Donc $T_{min//}(n) = s + \frac{1-s}{n}$. On obtient bien :

$$Acc_{max}(n) = T_{min//}(n)^{-1} = \left(s + \frac{1-s}{n}\right)^{-1}$$

□

2.3.3 Équilibrage de charge

Afin d'élaborer un algorithme parallèle efficace et offrant une bonne accélération, il y a une notion primordiale à aborder et à considérer, c'est la distribution de la charge entre les processeurs, et plus particulièrement l'équilibrage de charge.

Définition 2.14 *Le **partage de charge** a pour objectif de distribuer du travail aux unités de calcul disponibles.*

Certains traitements, typiquement un calcul non inter-dépendant entre les cases sur une matrice, présentent la particularité de fournir assez de travail en parallèle pour tous les processeurs de manière naturelle. Une simple politique de distribution de charge permet de faire travailler tous les processeurs pendant un temps relativement identique. Les processeurs ont donc tous une quantité équivalente de travail à effectuer, ce qui permet d'obtenir de ces processeurs un bon rendement, et de terminer les tâches parallèles à peu près dans le même temps. C'est un résultat idéal qui peut engendrer un algorithme performant. Toutefois, dans le cas d'un problème dont la résolution à l'étape i est dépendante des calculs à l'étape $i-1$, la parallélisation est plus difficile. SAT est dans ce cas, de plus, si on reprend la représentation en arbre binaire de la procédure DLL, il est impossible de prédire exactement le temps que prendra le calcul d'un sous-arbre. Ces points obligent l'informaticien qui souhaite paralléliser SAT à prendre en compte la problématique de l'équilibrage de charge.

Définition 2.15 *L'**équilibrage de charge** a pour objectif de donner du travail à toutes les unités de calcul disponibles, et ce, de manière équitable afin de minimiser les temps où les unités de calcul sont oisives.*

Exemple 26 *Pour SAT, il est possible de distribuer les sous-arbres droit et gauche de la racine sur 2 processeurs, c'est du partage de charge. Mais si le sous-arbre droit contient 10 fois plus de nœuds que celui de gauche, alors une stratégie d'équilibrage de charge tentera de subdiviser à nouveau le sous-arbre droit pour redonner du travail au processeur ayant déjà terminé l'exploration du sous-arbre gauche.*

Les éléments qui viennent d'être discutés seront approfondis pour la problématique SAT dans le chapitre 3.

Ce chapitre 2 ayant pour objectif de faire un état de l'art des modèles de parallélisme, nous commencerons par une présentation des modèles d'exécution. Ensuite, nous détaillerons les modèles d'architectures les plus utilisées pour résoudre SAT aujourd'hui ainsi que les modèles de programmation liés à ces architectures.

2.4 Modèles d'exécution, Taxinomie de Flynn

Dans un cadre séquentiel, il n'existe qu'un seul modèle d'exécution : une instruction à la fois sur un seul ensemble de données. C'est le modèle théorique de Von Neumann. En revanche, il existe différentes manières d'exécuter plusieurs tâches simultanément.

Michael J. Flynn proposa une classification [Fly66, Fly72, FR96] pour distinguer les modèles à instructions multiples ou simples et à données multiples ou simples. Il en découle quatre modèles distincts : SISD, SIMD, MISD et MIMD que nous allons développer de suite (voir figure 2.5).

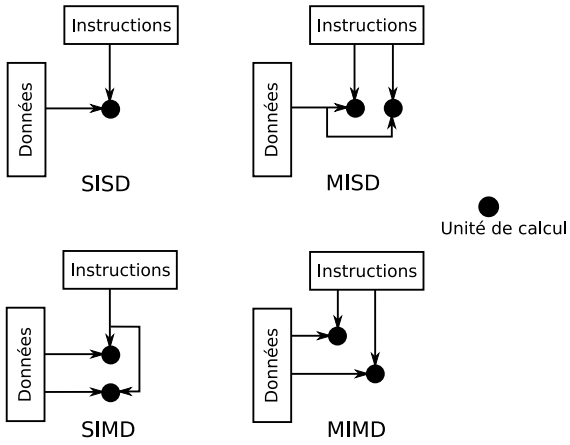


FIGURE 2.5 – Taxonomie de la classification de Flynn

2.4.1 SISD

Définition 2.16 ***SISD : Single Instruction Single Data****, machine pouvant traiter un seul flot d'instructions accompagné d'un seul flot de données.*

Modèle traditionnel du monde séquentiel, adapté aux architectures d'ordinateurs autour de processeurs scalaires et super scalaires.

2.4.2 SIMD

Définition 2.17 ***SIMD : Single Instruction Multiple Data****, machine constituée de plusieurs unités de calcul pouvant traiter un seul flot d'instructions accompagné de plusieurs flots de données.*

Ce modèle d'exécution synchrone entre les unités de calcul exécutent la même instruction au même instant mais sur des données différentes. Typiquement, ce sont les processeurs vectoriels capables de traiter un grand nombre de données sous forme de tableaux, chaque processeur traitant une sous partie du tableau. Ce type de parallélisme fonctionne bien lorsque le problème est uniforme puisque chaque processeur est en permanence alimenté par des données.

2.4.3 MISD

Définition 2.18 ***MISD : Multiple Instruction Single Data****, machine constituée de plusieurs unités de calcul pouvant traiter plusieurs flots d'instructions accompagnés d'un seul flot de données.*

Modèle très spécifique à certains traitements. Ce modèle s'exécute plus particulièrement sur des tableaux systoliques de processeurs : chaque processeur a une fonction propre et la donnée passe de processeur en processeur pour subir des traitements différents.

2.4.4 MIMD

Définition 2.19 ***MIMD : Multiple Instruction Multiple Data****, machine constituée de plusieurs unités de calcul pouvant traiter plusieurs flots d'instructions accompagnés de plusieurs flots de données.*

Ce modèle d'exécution est le plus répandu car le plus souple à utiliser. L'exécution d'un solveur SAT est très irrégulière en accès mémoire ; pour ces deux raisons, c'est ce modèle qui est le plus souvent utilisé par les chercheurs du domaine SAT. De cette manière il est possible de gérer différentes étapes de résolution à différents endroits dans l'espace de recherche. Le modèle MIMD peut être exécuté sur différentes architectures de machines selon la façon dont la mémoire est accédée par les processeurs. Nous allons maintenant vous définir ces différentes architectures.

2.5 Architectures matérielles relevant du modèle MIMD

Les modèles d'exécution MIMD ont plusieurs architectures cibles sur lesquelles s'exécuter. Johnson [Joh88] donna une classification selon la manière d'accéder à la mémoire. Il en résulte trois modèles d'adressage que nous allons développer ensuite :

- **CC-UMA** (*Cache Coherent - Uniform Memory Access*) : La mémoire est globale et partagée par tous les processeurs de l'ordinateur. Ils accèdent à la mémoire à la même vitesse.
- **CC-NUMA** (*Cache Coherent - Non Uniform Memory Access*) : Ici aussi, la mémoire est partagée et accessible par tous les processeurs mais la vitesse de transfert n'est pas la même. La mémoire locale est beaucoup plus rapide que la mémoire distante.
- **NoRMA** (*No Remote Memory Access*) : Dans cette architecture, la mémoire est distribuée sur les processeurs. Chaque processeur accède à sa mémoire locale mais ne peut accéder à celle des autres processeurs.

2.5.1 CC-UMA (*Cache Coherent - Uniform Memory Access*)

Cette architecture, parfois simplement appelée UMA, possède d'un côté la mémoire globale (souvent appelée « *RAM, Random Access Memory* ») et de l'autre plusieurs unités de calcul avec leurs mémoires caches (voir figure 2.6). Les unités de calcul (processeurs ou cœurs de calcul) sont reliés par un Bus d'interconnexion pour accéder à la mémoire et aux périphériques. Cet élément est donc partagé en concurrence pour accéder à la mémoire centrale, malheureusement cela limite la taille de ce type d'architectures à quelques dizaines d'unités de calcul.

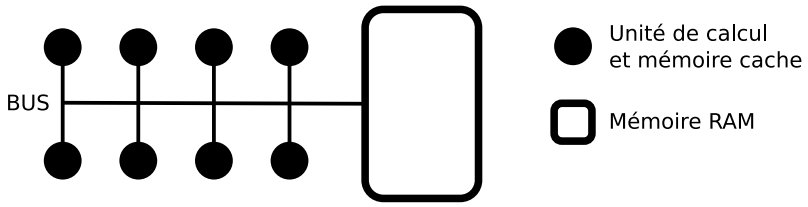


FIGURE 2.6 – Schéma d'architecture CC-UMA

Les mémoires caches servent à fournir les données aux unités de calcul en limitant les temps de latence car ces mémoires sont généralement cadencées à la même fréquence que l'unité de calcul. Les données ou le code à exécuter sont amenés de la mémoire globale à la mémoire cache de chaque cœur. La difficulté d'une architecture telle que CC-UMA est de gérer la cohérence des mémoires caches. En effet, c'est un réel problème à gérer et peut facilement se comprendre par un exemple simple (exemple 27).

Exemple 27 *Soit i une variable entière, posons $i = 0$ au début de l'algorithme. La donnée i est lue par les processeurs A et B pendant l'exécution de l'algorithme. L'information $i = 0$ est chargée dans la mémoire cache de A et dans celle de B. Supposons que A modifie i et lui affecte la valeur 25. Plus tard, si B souhaite réutiliser i pour un nouveau calcul, il doit avoir la valeur mise à jour par A plutôt que celle contenue dans sa mémoire cache sinon le calcul sera erroné. L'ordinateur doit donc être capable de gérer l'écriture de $i = 25$ en mémoire globale mais aussi dans les mémoires caches des processeurs qui ont déjà chargé i.*

Il est important que ce mécanisme se fasse de façon transparente pour le développeur car ce serait un véritable casse-tête à chaque développement d'algorithme parallèle.

Les machines du type CC-UMA sont appelées des SMP pour « *Symmetric Multi Processor* », machines multi processeurs symétriques. Typiquement, un ordinateur multi-cœurs est d'architecture CC-UMA.

2.5.2 CC-NUMA (*Cache Coherent - Non Uniform Memory Access*)

L'architecture CC-NUMA a la particularité de permettre l'adressage global de la mémoire, à la manière de l'architecture CC-UMA mais ici la mémoire partagée n'est pas connectée par un bus à l'ensemble des unités de calcul. Chaque processeur possède de la mémoire locale directement adressable à très grande vitesse, et adressable par les autres unités de calcul mais à vitesse réduite (voir figure 2.7). C'est de cette différence de vitesse que provient le caractère non uniforme de cette architecture. Le réseau d'interconnexion des unités de calcul peut être un bus, un hypercube, un arbre, ...

L'architecture CC-NUMA est aussi « *Cache-Coherent* », c'est à dire que la cohérence des mémoires cache est garantie par l'ordinateur. Le coût de cette cohérence mémoire est ici plus lourd que dans l'architecture CC-UMA puisque l'interconnexion est plus complexe et peut accueillir un plus grand nombre d'unités de calcul (de l'ordre de

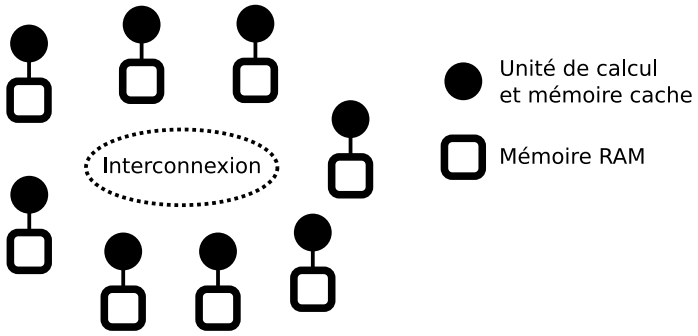


FIGURE 2.7 – Schéma d'architecture NUMA

plusieurs centaines). Par exemple, l'Altix 4700 de SGI permet de connecter 1024 cœurs de calcul en architecture CC-NUMA avec un total de 128 To de mémoire globale, la topologie de l'interconnexion est en *fat tree* (processeurs organisés en arbre mais la bande passante est de plus en plus large à mesure que l'on s'approche de la racine).

2.5.3 NoRMA (*No Remote Memory Access*)

Les ordinateurs à architecture NoRMA peuvent être schématisées par la figure 2.7 (comme pour les NUMA). En effet, les mémoires restent distribuées mais ici aucun système d'adressage global n'est géré. Il n'y a donc pas d'espace commun d'échange en mémoire, le partage d'information s'effectue de manière différente : l'échange de messages entre processeurs à travers le réseau d'interconnexion. Les difficultés de conception de ce type de machines sont moindres que les précédentes, par conséquent les ordinateurs ayant cette architecture peuvent compter plusieurs milliers d'unités de calcul.

2.5.4 Hybridation

Les super ordinateurs actuels sont en général constitués d'une hybridation des architectures présentées ci-dessus. En effet, il n'est pas rare de relier par une architecture NoRMA des nœuds de type CC-UMA ou CC-NUMA selon le nombre de processeurs. Les nœuds CC-NUMA étant eux-mêmes des grappes de CC-UMA. Cela permet de créer des machines extrêmement performantes en termes de puissance pure. Chaque nœud possède un nombre de processeurs non négligeable avec de hautes performances et pour les programmes devant s'exécuter sur un très grand nombre de processeurs, la machine procède par échange de messages.

D'après le site `www.top500.org` qui liste les 500 machines les plus puissantes au monde, la machine actuellement [3] au premier rang est Roadrunner. Cette machine a la particularité d'être constituée de deux types de processeurs. Composée de 8640 AMD Opteron contenant 2 cœurs et de 16560 Cell contenant 9 cœurs, Roadrunner affiche une puissance crête de 1,71 PFlops.

3. La plus récente liste à la date de rédaction de ce document est de juin 2009

En France, le calculateur le plus puissant est JADE du CINES (Centre Informatique National de l'Enseignement Supérieur), il est positionné à la $20^{ème}$ place du TOP500. C'est une machine NoRMA de 1536 nœuds, chacun étant un nœud CC-UMA constitué de 2 Intel Quad-Core E5472 (donc 8 cœurs par nœud). La topologie d'interconnexion est un hypercube. La puissance crête de JADE est de 147 TFlops.

Le super calculateur ROMEO II de l'Université de Reims Champagne-Ardenne entre aussi dans cette catégorie d'ordinateurs hybrides. Il est constitué de 96 processeurs dédiés au calcul hautes performances divisés sur 8 nœuds (les nœuds d'administration ne sont pas comptabilisés ici). Le tableau 2.1 donne les caractéristiques de chacun de ces nœuds. Les nœuds 8 et 9 comportant plus de 8 cœurs de calcul sont des nœuds de type CC-NUMA (8 cœurs en architecture CC-UMA reliés pour former une architecture CC-NUMA). Le réseau d'interconnexion entre les nœuds est de marque Quadrics (QS32A-CR 16 voies avec prise en charge MPI [4]) permettant une communication soutenue et bidirectionnelle de plus de 900 Mo/s entre les nœuds. Le système d'exploitation est basé sur une distribution GNU/Linux Red Hat retravaillée par Bull. La puissance crête de ROMEO II est de 614 GFlops.

Table 2.1 – Composition du calculateur ROMEO II

Nœud	Type	# Cœurs	Mémoire vive (Go)	Disque dur (Go)
romeo2	Novascale 3045	8	16	300
romeo3	Novascale 3045	8	16	300
romeo4	Novascale 3045	8	16	300
romeo5	Novascale 3045	8	16	300
romeo6	Novascale 3045	8	32	300
romeo7	Novascale 3045	8	32	300
romeo8	Novascale 5165	16	128	600
romeo9	Novascale 5325	32	64	600

2.5.5 Calculs distribués

Il est envisageable de connecter différentes machines par un réseau. Plutôt que de relier des processeurs au sein d'une même machine performante mais chère, il est parfois plus rentable de réutiliser de vieux ordinateurs pour les recycler en grappe (*cluster*) de PC. De cette manière, on obtient un réseau de machines très hétérogène, avec des connexions moins fiables, moins sûres et dont la vitesse n'est pas toujours la même. Les PC connectés peuvent même utiliser Internet comme médium. On obtient alors une architecture NoRMA sur le réseau. Dans ces conditions, on ne parle plus de calcul parallèle mais plutôt de calcul distribué. Le calcul sur grille (*Grid computing*) ou le calcul pair-à-pair (*peer-to-peer*) comme SETI@home sont des exemples de calculs distribués.

4. voir section 2.6.2

2.6 Modèles de programmation

Ce qui différencie un algorithme parallèle d'un algorithme séquentiel est le besoin de communication entre les processeurs. Nous allons présenter ici deux modèles conceptuels de communication pour écrire des algorithmes parallèles : le modèle PRAM (pour *Parallel Random Access Machine*) et le modèle DRAM (pour *Distributed Random Access Machine*) [CF91]. Dans le premier cas, les données sont écrites en mémoire et il suffit à chaque processeur qui requiert une information de la lire depuis cette mémoire. Dans le second cas, la communication s'effectue en échangeant explicitement des messages entre les processeurs.

2.6.1 Modèle PRAM

2.6.1.1 Présentation

Ce modèle ressemble beaucoup au modèle séquentiel appelé RAM (pour *Random Access Machine*). Dans ce modèle, il est fait l'hypothèse que le temps d'accès à la mémoire pour y lire une donnée de taille élémentaire est constant. De même, le temps d'effectuer une instruction de base est également faite en temps constant. D'un point de vue programmation, c'est le modèle le plus naturel puisqu'il n'est pas nécessaire de recourir à des moyens spécifiques pour accéder aux variables ou aux instructions. Les moyens de comparer les algorithmes écrits dans ce modèle sont la complexité en temps et la complexité en espace. Le modèle PRAM suggère de reprendre ce modèle mais en parallèle, c'est-à-dire par plusieurs processeurs simultanément. La mémoire reste accessible en temps constant pour l'ensemble des unités de calcul. Ce modèle suggère donc que la mémoire est directement accessible par l'ensemble des processeurs, or les accès concurrents à la mémoire est un dilemme. En effet, comment se comporte le modèle si différents processeurs accèdent simultanément à la même variable ? Il en résulte une classification des modèles PRAM en fonction de leur politique de gestion des accès à la mémoire :

- EREW (*Exclusive Read Exclusive Write*) : offre un accès exclusif à la mémoire, que ce soit en écriture ou en lecture. Le programmeur doit explicitement gérer des verrous d'accès pour s'assurer qu'un seul processeur accède à une variable.
- ERCW (*Exclusive Read Concurrent Write*) : accès exclusif en lecture mais concurrent en écriture, non utilisé, autant utiliser le modèle CRCW.
- CREW (*Concurrent Read Exclusive Write*) : Concurrence des accès en lecture autorisée car non problématique pour la cohérence des données. En revanche, l'accès en écriture se fait de manière exclusive. Le programmeur doit s'assurer que l'écriture ne s'effectue pas au même instant dans la même variable car le résultat serait indéterminé. C'est la politique standard, utilisée dans de nombreux algorithmes. Hormis la gestion des verrous pour garantir l'unicité d'écriture dans la mémoire, le modèle PRAM-CREW ressemble au modèle traditionnel et séquentiel RAM. La programmation en est donc que plus naturelle.
- CRCW (*Concurrent Read Concurrent Write*) : ici, les accès en lecture ou écriture sont concurrentiels. La lecture ne pose toujours pas de problème pour la cohérence

des données, mais ici, la politique d'écriture doit être déterminée à l'avance : ordre de priorité sur les processeurs, fonction de maximum ou minimum, ... Différentes stratégies peuvent exister et sont spécifiques à certains algorithmes.

Les architectures cibles les plus adaptées pour ce type de programmation sont les machines CC-UMA et CC-NUMA. Toutefois des compilateurs particuliers permettent de programmer en modèle PRAM sur des architectures NoRMA [KB03].

2.6.1.2 OpenMP et Pthreads

OpenMP et Pthreads (*threads* POSIX) sont des API (*Application Programming Interface*) permettant de produire du code exécutable en modèle PRAM. Avant d'expliquer plus précisément ces deux outils, il est nécessaire de succinctement définir la manière dont s'exécute un programme sur un ordinateur. Le programme est le code à exécuter enregistré sur le disque dur. Lorsque ce programme est lancé, il devient un processus dans l'ordinateur. Le processus se charge en mémoire et utilise des ressources telles que le processeur.

Définition 2.20 *Le **processus** (ou processus lourd) est composé d'une pile d'exécution qui contient les données lui permettant de savoir ce qui doit être exécuté ainsi que les valeurs des variables locales à une fonction. Il est aussi composé d'un espace mémoire lui permettant, par exemple, d'accéder aux variables globales du programme.*

Si le programmeur le souhaite, il peut faire appel à une sous catégorie de processus, les processus légers.

Définition 2.21 *Les **processus légers** (ou « threads », ce terme sera souvent utilisé) associés à un processus lourd partagent l'espace mémoire mais ont des piles d'exécution privées.*

Ces *threads* permettent d'adresser la même mémoire et donc les mêmes variables globales tout en permettant d'exécuter plusieurs suites d'instructions (fonctions) simultanément. Les API OpenMP et Pthreads implémentent ce type de technologie.

Les ***threads* POSIX** sont un standard d'implémentation des threads afin de rendre les applications multi-*threadées* portables sur différents systèmes. Les *threads* POSIX sont supportés sur l'ensemble des systèmes d'exploitation dérivés d'Unix. Ils se présentent sous la forme d'une bibliothèque de fonctions pour les langages C et C++. Les créations, verrous et synchronisations sont à gérer explicitement par le programmeur.

L'API **OpenMP** permet les mêmes gestions de créations, verrous, et synchronisations explicites que les *threads* POSIX, mais offrent en sus la possibilité de paralléliser automatiquement certaines parties du code source. Il suffit au programmeur d'annoter son code de commentaires spéciaux à destination du compilateur et ce dernier génère du code exécutable parallèle. Typiquement, pour effectuer un calcul parallèle sur une boucle « pour », il suffit de la précéder du commentaire « #pragma omp parallel for », le compilateur fera le reste. Il est toutefois nécessaire de préciser quelles sont les données privées des données à partager, mais la programmation parallèle peut être grandement

simplifiée par cette API. OpenMP est disponible pour les langages Fortran, C et C++. Il est nécessaire que le compilateur soit compatible, par exemple le compilateur d'Intel (ICC) et le compilateur GNU (GCC) le sont dans leurs récentes versions.

2.6.2 Modèle DRAM

2.6.2.1 Présentation

Le modèle DRAM est basé sur le principe de plusieurs unités de calcul indépendantes les unes des autres mais reliées entre elles. Chaque unité de calcul a accès en temps constant à sa propre mémoire locale. La seule manière de communiquer est de le faire explicitement : par l'échange de messages. Pour ce modèle particulier de programmation, il existe d'ailleurs un calcul de performances basé sur le nombre de messages échangés. Au même titre que l'on parle de complexité en temps et en espace dans les autres modèles, ici il existe la notion de complexité en messages. Cet échange engendre plusieurs inconvénients pour le développeur. Tout d'abord, le parallélisme est très explicite puisqu'il faut transmettre les données qui devront être traitées par d'autres unités de calcul, il faut donc savoir qui va les recevoir. Ensuite, il y a globalement un surcoût mémoire puisque les données doivent être envoyées puis réceptionnées et donc présentes plusieurs fois dans le système.

Ce modèle de programmation est principalement adapté aux architectures NoRMA mais peut aussi être exécutées sur des machines CC-UMA ou CC-NUMA. Dans ce dernier cas l'avantage de pouvoir adresser l'ensemble de la mémoire est perdu. En réalité, ce modèle de programmation a connu beaucoup de succès depuis le début du parallélisme car, pour des raisons de coût de fabrication, les premières machines massivement parallèles étaient de type NoRMA et ce modèle de programmation est l'adaptation algorithmique de ce modèle d'architecture.

2.6.2.2 MPI

La bibliothèque **MPI** (*Message Passing Interface*) permet d'échanger des messages entre processus. Elle est un standard développé par les constructeurs et éditeurs de logiciels depuis 1992 et permet donc la portabilité des codes. La version 2, de 1997, supporte les langages C, C++ et Fortran. Des versions payantes ou gratuites (par exemple MPICH) coexistent à l'heure actuelle.

2.6.3 Hybridation

À l'image des architectures qui peuvent être hybridées au sein d'une même machine, il est tout à fait envisageable d'hybrider les modèles de programmation au sein du même algorithme. Le but d'une telle manœuvre est bien entendu d'utiliser le maximum des performances des récents super calculateurs, quasiment tous basés sur une interconnexion NoRMA de nœuds CC-UMA ou CC-NUMA.

2.7 Conclusion

Ce chapitre avait pour objectif de présenter les motivations qui nous ont poussé à travailler sur le problème SAT dans un cadre parallèle. Ensuite, afin de donner au lecteur une vision globale des concepts et des technologies. Nous avons présenté les modèles d'exécution SISD, SIMD, MISD et MIMD. Puis nous avons présenté les différentes architectures liées au modèle MIMD : CC-UMA, CC-NUMA et NoRMA. Enfin, nous avons présenté les deux grands modèles de programmation PRAM et DRAM ainsi que leurs architectures de prédilection. Comme nous l'avons présenté en début de chapitre, la technologie multi-cœurs dans les micro-processeurs n'est pas le fruit d'un choix mais celui d'une limitation technique. Cela amène du parallélisme du type MIMD/CC-UMA dans tous les ordinateurs récents et ces machines permettent de développer dans un modèle PRAM, très naturel et très performant. Auparavant, ces ordinateurs étaient chers et complexes à concevoir mais ils deviennent maintenant la norme. Puisque cette technologie multi-cœur, peu gourmande en énergie et peu chère est récente, il n'existe pas beaucoup de solveurs SAT qui profitent de ces nouveaux ordinateurs puisque la plupart des solveurs SAT parallèles ont été développés dans un modèle DRAM. Le chapitre suivant présentera les difficultés de paralléliser un problème tel que SAT et dressera une liste non exhaustive des solveurs existants. Le solveur MTSS présenté dans ce document (chapitres 4 et 5) pose les bases d'une résolution de SAT complète dans un cadre multi-cœurs.

Chapitre 3

Résoudre SAT en parallèle

3.1 Introduction

Le problème SAT est très étudié dans un contexte séquentiel. Les progrès ont été nombreux et très importants, à tel point qu'il est parfois plus intéressant d'utiliser un solveur SAT plutôt qu'un algorithme dédié pour résoudre un problème voisin à SAT [TTKB09]. Compte tenu du développement actuel de la micro-informatique, il devient inéluctable que pour tirer le maximum de puissance de nos ordinateurs, il devient nécessaire d'élaborer des algorithmes parallèles. Néanmoins le parallélisme n'est pas un phénomène récent dans l'Informatique, il fut utilisé assez tôt par les militaires ou les scientifiques. Mais le parallélisme dans le domaine SAT n'est apparu que depuis le milieu des années 1990 [BS94], il y a une quinzaine d'années. C'est relativement jeune comparé aux bientôt cinquante années de résolution SAT en séquentiel depuis 1962 [DLL62]. Cette tardive implication des chercheurs en SAT dans le milieu du parallélisme est en particulier dûe à l'irrégularité de SAT, ce qui amène des problèmes de distribution des tâches vers les processeurs, ce point sera abordé dans ce chapitre, ainsi que la solution la plus utilisée pour y répondre : le chemin de guidage (*guiding path*). Ce chapitre abordera également différentes stratégies de parallélisme qui consistent à faire collaborer les processeurs (avec la problématique de l'équilibrage de charge) ou à les mettre en concurrence. Nous exposerons aussi les avantages que peuvent avoir les algorithmes parallèles sur les solveurs séquentiels, à savoir les gains super linéaires : dans certains cas, l'accélération parallèle est supérieure au nombre de processeurs utilisés. Nous finirons ce chapitre par une présentation des principaux solveurs SAT parallèles de l'histoire en modèle DRAM et nous ferons une présentation la plus exhaustive possible des solveurs en modèle PRAM. Tous ces solveurs sont ensuite présentés sous forme de tableau afin d'en effectuer un classement et une synthèse.

3.2 Difficultés

Il est en général assez simple de programmer un algorithme parallèle pour des algorithmes séquentiels traitant des matrices ou tableaux qui contiennent des données

non dépendantes les unes des autres. Cela pour deux raisons fondamentales : les données étant indépendantes, elles sont manipulables directement par les processeurs, et il est facile de donner autant de travail à chaque processeur (simple division de l'objet à travailler par le nombre d'unités de calcul disponibles).

Exemple 28 *Il est relativement simple de paralléliser automatiquement des boucles « pour ». L'API OpenMP est par exemple capable de faire ce genre de travail. Si le développeur fait un peu attention aux aspects mémoires des choses, il peut parfois avoir un parallélisme très efficace dans un tel cas de figure. La figure 3.1 montre la parallélisation d'un traitement systématique sur un tableau de 12 cases sur 3 processeurs. Puisque les tâches sont immédiatement disponibles et indépendantes entre elles, le parallélisme est ici très efficace : accélération du temps de calcul par 3 et donc une efficacité de 100% pour cet exemple.*

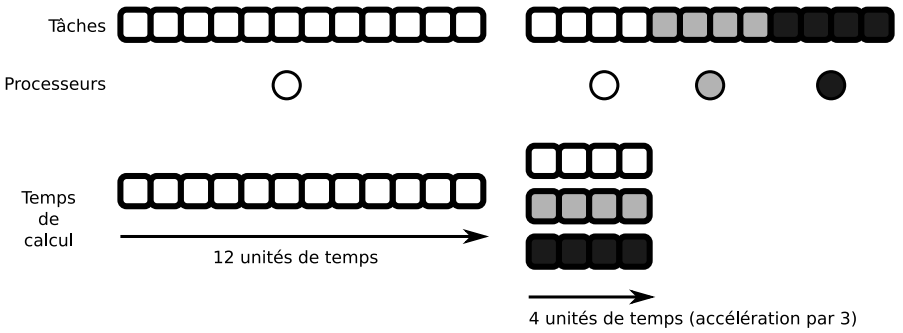


FIGURE 3.1 – Parallélisation d'un traitement sur un tableau

Le problème SAT n'est pas aussi facilement parallélisable car la représentation d'un tel processus de résolution peut s'apparenter à une recherche dans un arbre binaire. Chaque nœud dépend du calcul des nœuds précédents contenus dans son chemin jusqu'à la racine. Il est impossible de déterminer à l'avance la quantité de travail qui sera allouée à chaque processeur. Il y a donc deux problèmes à gérer pour que les processeurs aient toujours du travail : dégager assez de parallélisme et équilibrer dynamiquement la charge de travail entre les processeurs.

Exemple 29 *Prenons un exemple de 11 tâches parallélisables sur 3 processeurs avec une formule insatisfaisable. Elles sont dispersées dans un arbre binaire de recherche. L'algorithme schématisé dans la figure 3.2 assigne sur un processeur le sous-arbre immédiatement à droite après une division. Le déroulement décrit ici n'est qu'un exemple mais montre bien le caractère dépendant et non prévisible des tâches à accomplir. L'accélération en est réduite : seulement 1,57, ce qui correspond à une efficacité d'environ 50%.*

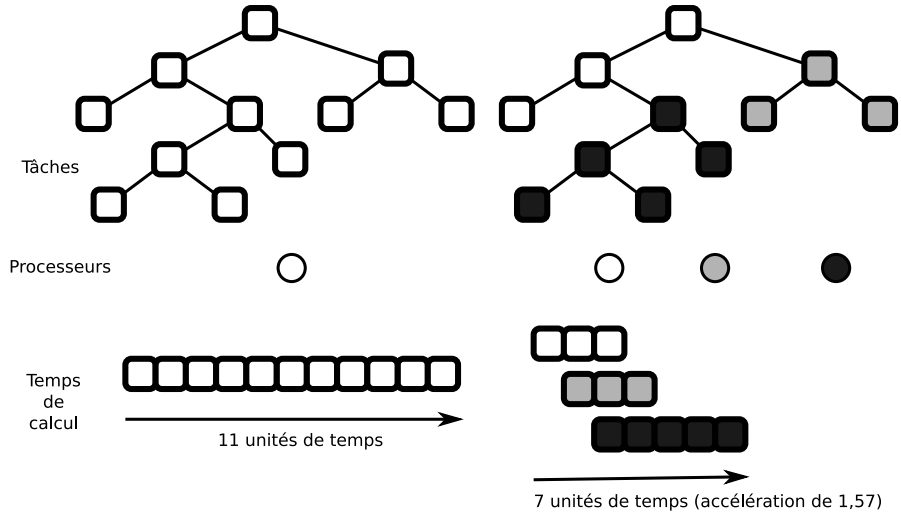


FIGURE 3.2 – Parallélisation d'une résolution SAT

3.3 Dégager du parallélisme

Il y a deux manières de faire travailler des processeurs simultanément : de façon collaborative ou de manière concurrente. En coopération, les processeurs travaillent ensemble sur la résolution du problème, ils se partagent le travail. Mais les processeurs peuvent être mis en concurrence pour travailler sur le même problème dans différentes stratégies de résolution avec des paramétrages différents. Le premier qui finit son travail pourra faire stopper les autres processeurs. Voici ces approches détaillées.

3.3.1 Aspects concurrents

La mise en concurrence de processus de résolution d'un problème SAT pour gagner du temps en parallèle n'a de sens que pour certains types d'algorithmes, puisque par exemple les algorithmes non paramétrables ou qui n'ont pas de comportement aléatoire ne pourront pas procéder à différentes recherches de solutions. Si nous disposons de n processeurs, les n processeurs feront n fois le même travail. Il n'y a rien à gagner. À l'inverse, les algorithmes du type incomplet, basés sur une recherche locale pourront tirer partie de l'aspect concurrentiel sur plusieurs processeurs. En lançant n processus de résolution sur n processeurs avec n graines aléatoires différentes, nous obtiendrons n recherches de solution différentes et le parallélisme pourra apporter un gain, non certifié *a priori* mais probable en moyenne.

Exemple 30 *Soit A un algorithme de recherche locale. Son exécution est limitée à* 50000 *inversions de valeurs de variables et à* 1000 *essais. Si on exécute A sur un ensemble de* 8 *processeurs avec* 8 *graines aléatoires différentes, cela correspond au lancement de A sur un seul processeur avec toujours* 50000 *inversions possibles mais avec* $1000 \times 8 = 8000$ *essais. L'avantage est que l'on reste dans un temps de calcul, dans le*

pire des cas, à 1000 *essais. Cela dit, rien ne garantit que le temps de résolution sera plus petit puisqu'une exécution séquentielle peut trouver une solution au bout de 200 essais, là où la version parallèle peut ne pas en trouver en 8000 essais.*

Une autre famille de solveurs se prête assez bien à la parallélisation en concurrence, ce sont les solveurs dits « modernes » : les solveurs récents destinés à résoudre les formules industrielles. Ces solveurs sont assez sensibles à leur paramétrage en fonction de la formule à résoudre. Un lancement en concurrence permet de résoudre une formule avec différentes stratégies, par exemple la proportion de choix aléatoires dans l'heuristique ou une politique de redémarrage plus ou moins agressive. Ces différentes stratégies exécutées en parallèle tentent d'amener de la robustesse dans ce type de solveur, c'est-à-dire que les performances sont plus homogènes quelque soit la formule à résoudre que la version séquentielle.

Les solveurs modernes développent des arbres de recherche, il est donc possible de faire travailler plusieurs processeurs de manière collaborative sur cet arbre mais les *backtracks* non chronologiques permettent de remonter plusieurs niveaux dans l'arbre en une seule étape, et cela constitue une autre raison pour laquelle le travail collaboratif présente des inconvénients. Il est possible qu'un sous-arbre qui n'aurait jamais été développé en séquentiel, le soit en parallèle ; un tel phénomène est montré en figure 3.3. Le même principe de nœuds développés en vain peut se retrouver dans le cadre des redémarrages effectués par ces solveurs. Par exemple si un seul processeur gère les redémarrages, les autres développeront quelques nœuds inutilement, ce travail sera perdu.

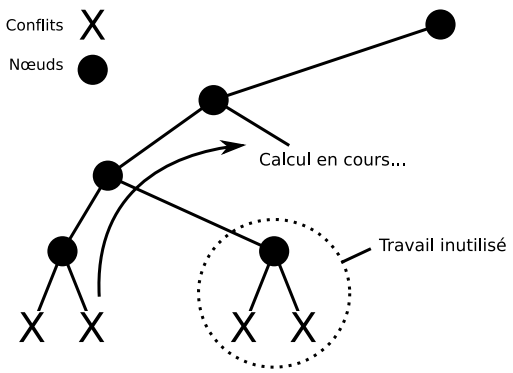


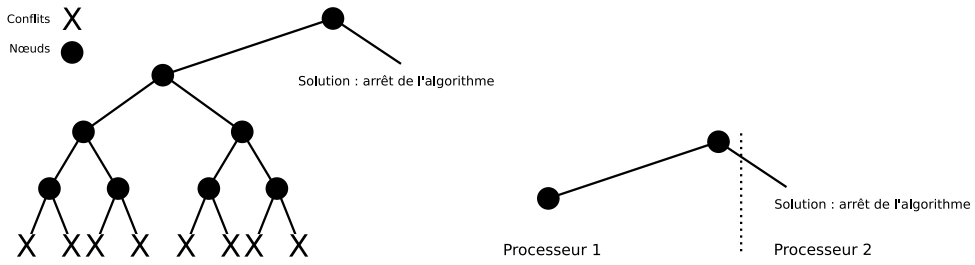
FIGURE 3.3 – *Backtracks* non chronologiques et travail inutile

Contrairement à la coopération de processeurs, il n'est pas nécessaire d'effectuer un équilibre de la charge lorsque les processeurs sont mis en concurrence. Les processeurs seront toujours utilisés à 100% et le premier à trouver une réponse enverra un ordre d'arrêt aux autres processeurs.

3.3.2 Aspects coopératifs

Pour dégager du travail parallèle pendant la résolution de SAT tout en faisant travailler les processeurs sur le même problème, il est possible de le faire à plusieurs niveaux. L'arbre de recherche peut être décomposé en différentes tâches plus ou moins importantes (nœuds ou sous-arbres), on parle alors de parallélisme par décomposition de domaine (*domain decomposition*). Il est aussi possible de paralléliser le traitement effectué à chaque nœud, dans ce cas, on parle de parallélisme bas niveau (*low level*). Plusieurs raisons font qu'il est préférable de diviser l'arbre de recherche pour paralléliser efficacement SAT plutôt que de paralléliser le traitement à chaque nœud.

- SAT étant un problème de décision, l'algorithme s'arrête dès la découverte d'une solution. Dans le cas de la résolution d'une formule satisfaisable, une accélération super linéaire est possible comme le montre la figure 3.4. Le nombre de tâches est considérablement réduit permettant de telles accélérations [SMV87].
- Le traitement effectué à chaque nœud de l'arbre n'est pas uniforme, beaucoup de données sont mises à jour mais ne sont généralement pas contigües en mémoire et ne sont pas connues à l'avance. Le parallélisme ne serait pas si performant que celui de la figure 3.1 et pas si simple à mettre en œuvre.
- Paralléliser les traitements locaux aux nœuds est un travail à fournir pour chaque heuristique ou pour chaque amélioration à apporter à l'algorithme. Paralléliser la distribution des nœuds ou des sous-arbres permet de s'abstraire de ce travail et permet d'utiliser n'importe quelle heuristique ou amélioration séquentielle sans un surplus de programmation.



(a) Résolution séquentielle (b) Résolution parallèle

FIGURE 3.4 – Accélération super linéaire dans le cas de formules satisfaisables

La meilleure solution pour résoudre SAT coopérativement est donc bien la division de l'arbre binaire de recherche. La distribution de sous-arbres à calculer correspond à un gros grain de parallélisme alors que la parallélisation des nœuds est un grain très fin. Cette approche amène cependant un problème puisque la taille des sous-arbres n'est pas connue à l'avance, il est nécessaire de mettre en place une politique d'équilibrage de charge dynamique pendant la résolution. Ceci est expliqué en section 3.4.

3.4 Équilibrage de charge dynamique

Comme nous venons de le décrire, dans le cadre d'un travail coopératif entre les processeurs, il est primordial de procéder à un équilibrage de charge dynamique tout au long de la résolution de SAT.

3.4.1 Présentation des équilibrages de charge

L'équilibrage de charge [LM82] dynamique est tout d'abord à opposer à l'équilibrage de charge statique. Ce dernier est assez naturel puisqu'il correspond à une distribution des tâches à calculer sur les processeurs avant que le processus de résolution ne débute. En SAT, un tel partage n'est pas possible puisque nous ne connaissons pas à l'avance la physionomie de l'arbre de recherche. Toutefois, Böhm et Speckenmeyer ont introduit un solveur en 1994 [BS94] qui tente d'estimer la taille des sous-arbres qu'il va développer pour organiser son équilibrage de charge.

Différentes catégories d'équilibrage de charge dynamique sont à distinguer selon qu'il est géré de manière centralisée ou distribuée, et selon que ce sont les processeurs oisifs ou ceux surchargés qui initient l'équilibrage.

3.4.1.1 Modèle centralisé

Deux types de processeurs sont distingués : le maître et les esclaves.

Définition 3.1 *Un processeur **maître** est chargé de distribuer les tâches aux esclaves.*

Définition 3.2 *Un processeur **esclave** est chargé de calculer une tâche.*

Dans le modèle centralisé un processeur est différencié, le maître. Il a l'unique charge de gérer l'équilibrage de charge des esclaves. Encore une fois deux manières de procéder sont à distinguer.

- **Liste centralisée** : le maître tient à jour localement, dans sa mémoire, une liste de tâches (le terme « tâche » est volontairement général, ce peut être des nœuds ou des sous-arbres) à disposition des esclaves. Lorsqu'un esclave est oisif, il demande au maître une nouvelle tâche.
- **Liste distribuée** : chaque esclave tient localement une liste de tâches à traiter et le maître a la charge de mettre en relation un esclave oisif et un esclave surchargé pour qu'ils communiquent afin que l'esclave oisif reçoive au moins une tâche de l'autre processeur esclave.

3.4.1.2 Modèle distribué

Deux types de processeurs sont distingués [WM85] : les serveurs et les sources.

Définition 3.3 *Un processeur **source** envoie des tâches aux serveurs.*

Définition 3.4 *Un processeur **serveur** est chargé de calculer une tâche.*

Il est important de noter qu'ici, les processeurs sont en général sources et serveurs à la fois. Cette distinction se fait dans le temps, une partie du temps est dédiée au calcul et une autre est dédiée à l'équilibre de charge.

Deux stratégies existent :

- **Source initiative** : se dit d'une stratégie qui charge les processeurs sources, déjà surchargés de tâches, de transmettre aux processeurs serveurs de nouvelles tâches. Les sources allègent donc leur travail alors que les serveurs peuvent reprendre leurs calculs. Le choix du serveur à qui le source envoie une ou plusieurs tâches peut être aléatoire, aléatoire avec seuil (pas trop de tâches sur le serveur) ou après élection de celui ayant le moins de tâches parmi un groupe de serveurs [ELZ86].
- **Serveur initiative** : se dit d'une stratégie qui repose sur le principe que les processeurs serveurs oisifs prennent l'initiative de récupérer des tâches depuis les serveurs sources surchargés.

3.4.2 Chemin de guidage

Parmi les solveurs parallèles complets existants, la méthode la plus utilisée pour distribuer les tâches de calcul est le chemin de guidage (*guiding path*) introduit par le solveur `PSATO` [ZB94, ZBP+96]. Le grain parallèle est gros puisque ce sont des sous-arbres qui sont partagés. Le chemin de guidage est un objet permettant d'indiquer sur quel sous-arbre un processeur doit travailler, mais c'est un aussi un objet qui indique les sous-arbres disponibles en parallèle. Un chemin de guidage est représenté par une suite de couple $< L_i, \delta_i >$, où L_i est un littéral à propager et δ_i indique si une seule des branches est en cours de calcul ou si la deuxième l'est aussi (la première étant en cours de calcul ou terminée). Une représentation d'un chemin de guidage est donnée en figure 3.5.

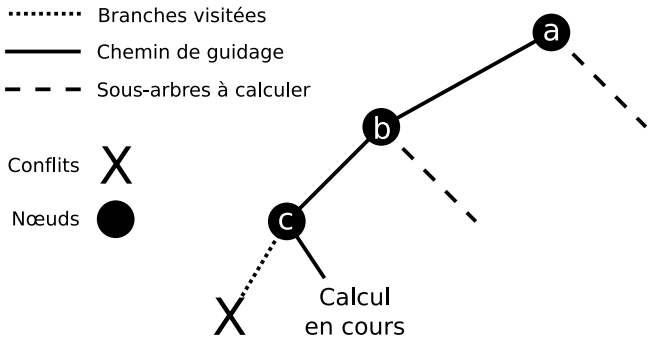


FIGURE 3.5 – Chemin de guidage

Si on choisit de symboliser un nœud dont les deux sous-arbres sont traités par 0 et par 1 le cas où un sous-arbre n'est pas encore calculé, alors le chemin de guidage de la figure 3.5 est $(< a, 1 >< b, 1 >< c, 0 >)$. Les sous-arbres non développés, pendants aux nœuds du chemin de guidage, sont des tâches qui sont assignables à des processeurs qui débuteront alors une recherche de solution dans les sous-arbres représentés par les

chemins. Les chemins de guidage sont utilisés pour connaître les sous-arbres qui doivent être explorés, mais aussi pour assigner du travail à un processeur. En effet, par exemple, à partir du chemin de guidage $(< a, 1 >< b, 1 >< c, 0 >)$, il est possible de construire deux chemins de guidage différents qui permettent d'orienter deux processeurs : $(< \bar{a}, 0 >)$ et $(< a, 0 >< b, 1 >< c, 0 >)$. Il est possible d'appliquer le même raisonnement sur la variable b, et ainsi obtenir les 3 chemins que l'exemple de la figure 3.5 est capable de fournir.

3.4.3 Équilibrage de charge par chemins de guidage

Le modèle mis en place pour équilibrer la charge par une distribution de chemins de guidage est généralement centralisé à liste centralisée au sein du processeur maître. Le maître maintient à jour une liste de chemins de guidage. Un chemin permet de symboliser le futur travail sur un sous-arbre puisque le chemin permet de retrouver la racine d'un sous-arbre.

3.4.3.1 Fournir du travail

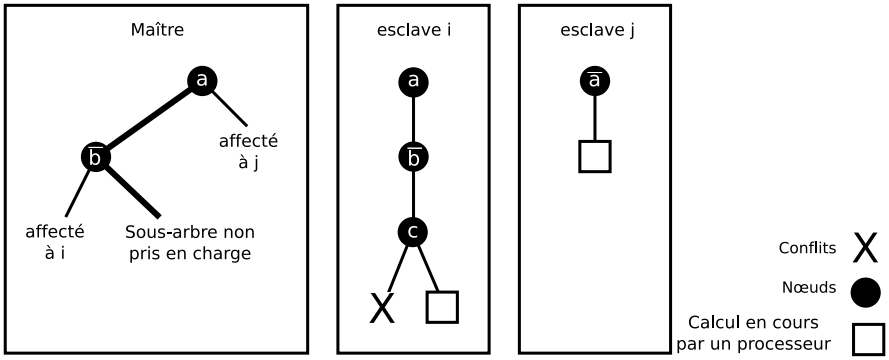
Au début de l'algorithme, chaque esclave doit connaître la formule d'origine, après modifications si la formule a été pré-traitée séquentiellement. L'important est seulement que chaque esclave reçoit une formule logique équivalente, appelons la $\mathcal{F}$. Par exemple, dans `PSATO`, les esclaves reçoivent à chaque fois le nom du fichier contenant la formule de départ à résoudre. Lorsqu'un esclave, i, manque de travail, il s'adresse au maître (processeur qui doit être identifié et connu de tous), afin de recevoir un nouveau chemin de guidage P. P est une suite de littéraux à propager dans la formule pour calculer une sous-formule $\mathcal{F}\backslash P$. À partir de cette sous-formule, i peut reprendre son travail de recherche de solution. Ceci est schématisé par la figure 3.6.

3.4.3.2 Maintenir la liste des tâches

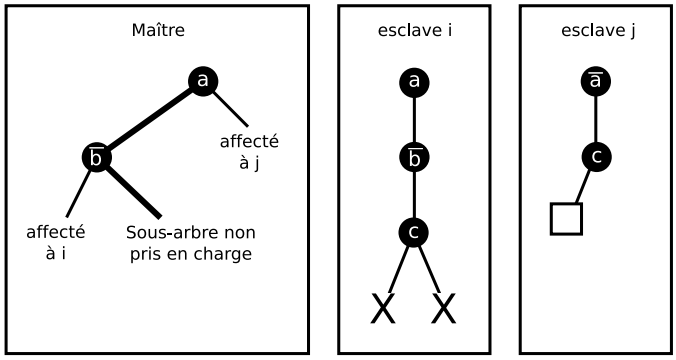
Le processeur maître doit absolument maintenir de nombreux chemins de guidage afin de pouvoir donner du travail dès qu'un esclave en fait la demande. Le maître doit régulièrement stopper des esclaves pour récupérer leurs chemins de guidage courants respectifs. De ces chemins, il détecte chaque sous-arbre que l'esclave n'a pas encore traité et il les conserve dans sa liste de tâches, ce sont autant de chemins de guidage disponibles. Le principe est donné par la figure 3.7, dans la réalité, le maître n'attend pas que sa liste soit vide pour récupérer des chemins non explorés, par exemple dans `PSATO`, le maître maintient un minimum de 10% de tâches supplémentaires que le nombre de processeurs esclaves.

3.4.3.3 Arrêt de l'algorithme

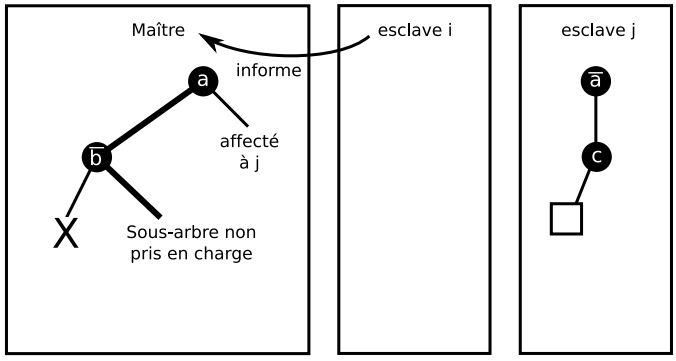
Si une formule est satisfaisable, un esclave trouvera une solution pendant son processus de résolution. Il lui suffira d'en avertir le processeur maître et ce dernier pourra stopper l'ensemble des processeurs esclaves. Dans le cas d'une formule insatisfaisable,



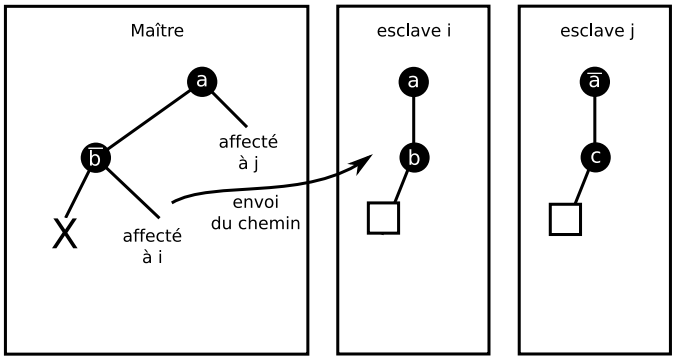
(a) état initial



(b) i a terminé son sous-arbre

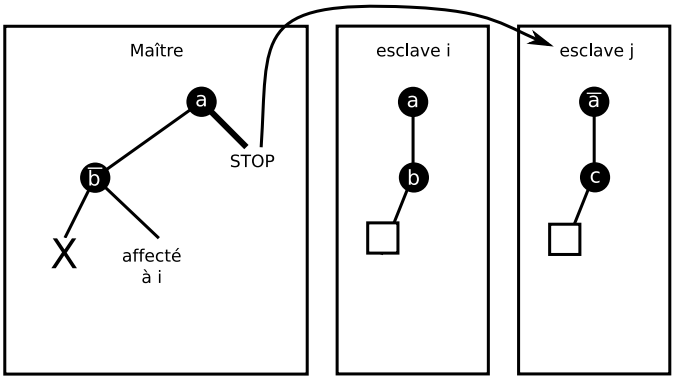


(c) i informe le maître

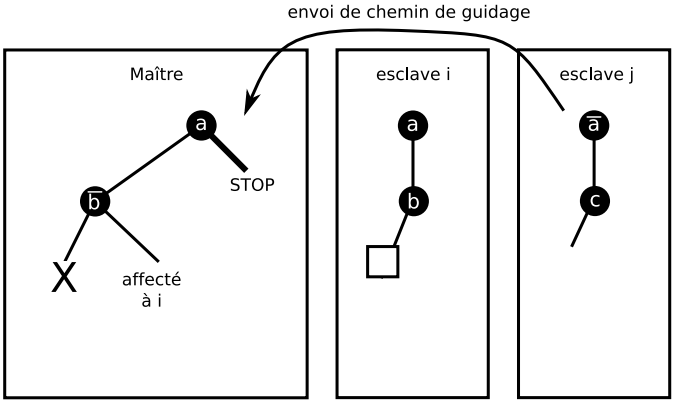


(d) i récupère un chemin de guidage depuis le maître

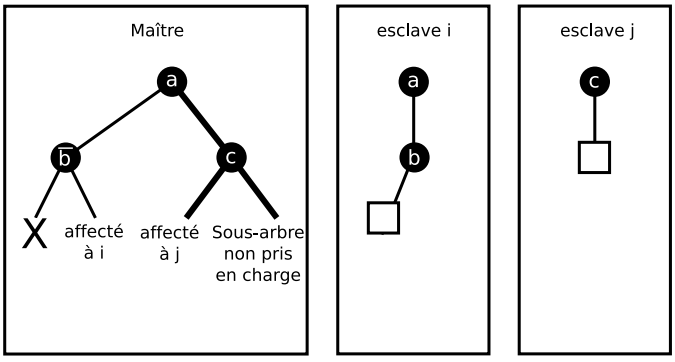
FIGURE 3.6 – Le maître distribue du travail aux esclaves



(a) Le maître n'a plus de chemins à fournir, il stoppe j



(b) j envoie son chemin de guidage



(c) j devra demander un chemin pour *backtracker*

FIGURE 3.7 – Le maître récupère ses chemins depuis les esclaves

les esclaves, n'ayant qu'une vue partielle de la résolution ne pourront rien détecter, ils demanderont donc du travail au maître. Ce dernier pourra conclure sur l'insatisfaisabilité de la formule puisque tous ses esclaves seront oisifs et lui n'aura aucune tâche à leur faire exécuter.

3.4.3.4 Intérêt

Le principe du chemin de guidage est très lié au modèle de programmation utilisé pour s'adapter aux machines parallèles les plus répandues à la fin des années 90 : le modèle DRAM pour architectures NoRMA (ou même pour les calculs distribués). En effet, la taille des données échangées est un critère primordial pour obtenir de bonnes performances, de même pour le nombre d'échanges, plus ces deux critères sont petits, plus un algorithme de type DRAM peut espérer avoir de bonnes performances. Or, le chemin de guidage répond bien à ces critères car les sous-arbres peuvent représenter un temps de calcul non négligeable, les esclaves peuvent être occupés longtemps sans revenir souvent vers le maître pour demander des sous-arbres. Néanmoins ce phénomène peut se produire lorsque des processeurs obtiennent des sous-arbres proches des feuilles de l'arbre binaire de recherche. Ce phénomène est nommé le phénomène de *ping-pong* et a été montré par le solveur `//satz` [JLU01]. Deuxième avantage, le chemin ne représente pas beaucoup de données à transmettre : seulement quelques entiers.

3.5 Échange de lemmes

Partager du travail est une chose, mais partager de l'information en est une autre. Il faut au préalable un algorithme qui sache extraire de l'information du problème mais aussi que cette information ne soit pas trop conséquente pour ne pas écrouler les performances. C'est ce que permet dans une certaine limite l'échange de clauses. Cet échange est tout aussi possible dans un cadre coopératif que dans un cadre concurrentiel.

3.5.1 Apprentissage de clauses

La résolution des formules industrielles est grandement accélérée par les solveurs modernes capables d'apprendre des clauses à partir des conflits rencontrés. La raison de cette amélioration est simple : les clauses apprises permettent de réduire le nombre de nœuds dans l'arbre de recherche comme le montre les performances des solveurs modernes. Le travail diminue. Cet échange de clauses est possible sous la forme de messages échangés pour les algorithmes basés sur le modèle DRAM, ou sous la forme d'une base de clauses partagée pour les algorithmes fondés sur le modèle PRAM. Ces échanges entre processeurs permettent de gagner du temps par la réduction du travail, c'est indéniable, mais le plus intéressant est la possibilité d'obtenir des gains super linéaires même pour des formules insatisfaisables.

Preuve: Soient $\mathcal{A}$ et $\mathcal{B}$ deux sous-arbres ne contenant pas de solution pour une formule $\mathcal{F}$. Soient $n_{\mathcal{A}}$ et $n_{\mathcal{B}}$ les nombres de nœuds développés dans $\mathcal{A}$ et $\mathcal{B}$ pendant

une résolution séquentielle sans conservation des clauses apprises. Soient $C_{\mathcal{A}}$ et $C_{\mathcal{B}}$ les clauses apprises après avoir fini de développer $\mathcal{A}$ et $\mathcal{B}$.

Prenons maintenant une résolution séquentielle qui conserve les clauses apprises pour changer de sous-arbre. Admettons que $\mathcal{A}$ soit développé avant $\mathcal{B}$, $C_{\mathcal{A}}$ est ajouté à la base de clauses avant de résoudre $\mathcal{B}$. On obtient $n'_{\mathcal{B}}$ le nombre de nœuds développés pour résoudre $\mathcal{B}$. L'apprentissage ne peut que réduire le nombre de nœuds et non l'augmenter, on a donc $n'_{\mathcal{B}} \leq n_{\mathcal{B}}$. Inversons l'ordre de résolution de $\mathcal{A}$ et $\mathcal{B}$, on obtient avec un raisonnement similaire un nombre de nœuds $n'_{\mathcal{A}}$ tel que $n'_{\mathcal{A}} \leq n_{\mathcal{A}}$ grâce aux clauses de $C_{\mathcal{B}}$ ajoutées à la formule.

Mettons-nous maintenant dans un cadre parallèle où $\mathcal{A}$ et $\mathcal{B}$ sont calculés simultanément. On ne peut pas affirmer que $\mathcal{A}$ et $\mathcal{B}$ seront résolus en $n'_{\mathcal{A}}$ et $n'_{\mathcal{B}}$ puisque ces nombres sont obtenus avec la présence respectives de $C_{\mathcal{B}}$ et $C_{\mathcal{A}}$ avant le début de la résolution, or ici elles se font simultanément. À tout instant de la résolution de $\mathcal{A}$ (resp. $\mathcal{B}$), seul un sous-ensemble de $C_{\mathcal{B}}$ (resp. $C_{\mathcal{A}}$) permet de réduire la formule. Ainsi, on obtient les nombres de nœuds développés $n''_{\mathcal{A}}$ et $n''_{\mathcal{B}}$ pour résoudre $\mathcal{A}$ et $\mathcal{B}$ en parallèle. Cependant, il est certain que $n'_{\mathcal{A}} \leq n''_{\mathcal{A}} \leq n_{\mathcal{A}}$ et $n'_{\mathcal{B}} \leq n''_{\mathcal{B}} \leq n_{\mathcal{B}}$. Il est alors envisageable, mais non garanti, que $n''_{\mathcal{A}} + n''_{\mathcal{B}} \leq n_{\mathcal{A}} + n'_{\mathcal{B}}$ (si $\mathcal{A}$ est développé avant $\mathcal{B}$), offrant par la même occasion des accélérations super linéaires à la résolution de $\mathcal{F}$.

□

Cette preuve ne démontre pas que des gains super linéaires sont systématiques, mais seulement qu'ils sont possibles. Ils sont à contre-balancer par les surcoûts induits par le parallélisme (équilibrage de charge, échange des clauses, ...). En réalité, il est nécessaire d'avoir un écart assez grand entre $n''_{\mathcal{A}} + n''_{\mathcal{B}}$ et $n_{\mathcal{A}} + n'_{\mathcal{B}}$ pour espérer recouvrir les surcoûts. Comme nous le verrons au chapitre 6, de telles accélérations sont tout de même observables en pratique.

3.5.2 Limitations pratiques

3.5.2.1 Modèle DRAM

Ce modèle requiert toujours de minimiser les échanges de messages afin de ne pas effondrer les performances à cause d'une surcharge de l'élément réseau. Le problème de l'apprentissage (dans un cadre séquentiel aussi mais dans une moindre mesure) est que le nombre de clauses pouvant être apprises est très grand : à chaque conflit plusieurs clauses sont ajoutées à la formule. Ce nombre de conflit est au maximum de 2^n avec n variables, il est donc impossible de tout apprendre. Comme expliqué en section 1.9.1, les algorithmes séquentiels effacent régulièrement les clauses les moins utilisées. Dans un cadre parallèle, l'idéal est de minimiser les clauses échangées inutilement. Un critère simple est de partager les clauses les plus petites possibles. Elles permettent de réduire la taille des messages à échanger, mais ce sont aussi les plus utiles car elles sont rapidement réduites et provoquent donc plus souvent des propagations unitaires.

3.5.2.2 Modèle PRAM

Même si la longueur des clauses ne provoquent pas ici de lourds échanges, il est nécessaire de poser des verrous pour l'écriture d'une clause dans la base de données. Cela conduit à une séquentialisation de l'écriture des clauses. Le fait de réduire le nombre de clauses partagées entre les processeurs amène de meilleures performances de la base de données. Là encore, on peut procéder à une sélection par la taille. Quelques améliorations spécifiques au modèle PRAM sont cependant possibles. Par exemple, dans [LSB07] pour le solveur `MiraXT`, les chercheurs ont mis en place une base partagée avec mise à jour ultra rapide. Il est toujours nécessaire d'utiliser un verrou pour la mise à jour mais cette dernière est réduite au maximum : ajout d'un pointeur vers la clause déjà écrite par le *thread* qui l'a apprise puis incrémentation du pointeur de la base partagée. L'utilisation de pointeurs depuis la base partagée vers les bases des *threads* permet en sus de ne pas avoir redondance d'écriture en mémoire des clauses partagées, toute clause n'est présente qu'une seule fois. La suppression d'une clause est laissée à la charge du *thread* qui l'a apprise, ainsi la base partagée n'a pas besoin d'être verrouillée pendant cette étape. Lorsque beaucoup de suppressions ont eu lieu, la base de clauses partagée se trouve morcelée, l'algorithme décide donc de temps en temps, mais pas systématiquement, de verrouiller la totalité de la base afin d'effectuer le nettoyage de celle-ci.

3.6 Présentation de solveurs SAT parallèles

3.6.1 Solveurs DRAM

Voici une présentation succincte de quelques solveurs SAT parallèles à échange de messages, donc plutôt destinés à être exécutés sur des machines d'architecture NoRMA.

3.6.1.1 Solveur de Max Böhm et Ewald Speckenmeyer

Premier véritable solveur parallèle de l'histoire en 1994 [BS94], il a la particularité de présenter un très bon passage à l'échelle puisque son efficacité est d'environ 95% sur 256 processeurs. Les chercheurs préconisent une utilisation de machines à processeurs interconnectés en grille en cas d'utilisation de plus de 256 processeurs. Chaque processeur exécute 2 fonctions distinctes : le travailleur et l'équilibreur. Le travailleur traite le sous-arbre de la formule à résoudre par un des plus performants solveur de l'époque tandis que l'équilibreur estime le temps de calcul d'un tel sous-arbre. L'équilibreur tente de garder en permanence au moins 3 tâches à traiter en réserve afin que le travailleur ne soit jamais inutilisé. Sinon, il requiert une phase d'équilibrage. Ainsi, ce solveur est totalement distribué, sans recours à un modèle maître/esclave. Ce solveur était utilisé et optimisé pour résoudre des formules k-SAT aléatoires.

3.6.1.2 PSATO

Lui aussi est un des pionniers de la résolution de SAT en parallèle [ZB94, ZBP+96] (1994). Ce solveur a introduit la notion simple et efficace de chemin de guidage. Il est basé sur le solveur séquentiel SATO [Zha93]. Une explication plus approfondie du chemin de guidage est donnée en section 3.4.2, mais pour résumer PSATO, un processeur maître est chargé de distribuer des chemins qui permettent aux processeurs esclaves de savoir sur quels sous-arbres ils doivent travailler. Parfois, lorsque le processeur maître estime qu'il n'a pas assez de chemins en réserve, il stoppe l'exécution d'un ou plusieurs esclaves pour lui (leur) demander de lui retourner son chemin courant. Tous les sous-arbres non encore explorés de ce(s) chemin(s) feront autant de futures tâches pour l'ensemble des processeurs esclaves. PSATO est spécialisé dans la résolution de formules aléatoires.

3.6.1.3 //satz

Version parallèle [JLU01] du solveur nommé satz [LA97]. Ce solveur séquentiel (satz) intègre l'heuristique UP destinée à maximiser le nombre de propagations unitaires pendant la recherche. Le but de cette manœuvre est de minimiser le nombre de choix possibles pour réduire la taille de l'arbre de recherche. Il existe en réalité deux versions parallèles de //satz, en effet une version dédiée au calcul sur 2 processeurs a été créée sans utilisation de processeur maître spécialisé dans l'équilibrage de charge. L'autre version peut être utilisée sur n processeurs, mais dans ce cas, un processeur est dédié à l'équilibrage de charge. Les chercheurs ont mis en évidence un phénomène de *ping-pong* dû à l'utilisation du chemin de guidage. Ce phénomène résulte d'un accroissement du nombre d'échanges de chemins entre le maître et les esclaves au fur et à mesure que les esclaves calculent des sous-arbres proches des feuilles. //satz est spécialisé dans la résolution de formules aléatoires.

3.6.1.4 PaSAT

Premier solveur parallèle à échanger des clauses en plus du traditionnel chemin de guidage [SBK01, BSK03]. C'est donc un solveur destiné à résoudre les formules industrielles. Il a été le premier confronté aux problèmes liés à ce type d'échanges, à savoir le nombre et la taille. En effet, une clause est apprise à chaque feuille de l'arbre et le nombre de feuilles pendant la résolution de SAT est exponentiel par rapport au nombre de variables dans la formule. Ensuite, la taille des clauses peut être très grande et donc représenter une communication relativement importante. Il est nécessaire d'appliquer une politique de discrimination des clauses échangées, et le critère qui sera choisi pour PaSAT est la taille de la clause. En pratique, ce critère sera repris par tous les autres solveurs industriels parallèles.

3.6.1.5 GridSat

Ce solveur distribué est conçu pour le calcul sur grille de PC [CW06]. Sa philosophie est de conserver au maximum une exécution séquentielle et de ne paralléliser les tâches

que lorsque le solveur estime cela avantageux. Le solveur SAT lancé par les clients est `zchaff`, un solveur industriel performant. Le processeur maître distribue le travail parmi les esclaves et tient à jour une base de données distribuée des clauses apprises.

3.6.1.6 `PSolver`

Solveur distribué pour calcul pair à pair [Kok98]. Le solveur utilisé sur les machines cibles n'est pas fixé à l'avance et les tâches sont distribuées par des chemins de guidage aux hôtes. Ici, c'est un serveur qui fait office de maître pour sauvegarder les chemins de guidage. Ce solveur doit faire face à de nombreuses déconnexions et reconnexions de ses hôtes. Malgré une approche originale, ce solveur n'a jamais fait l'objet de résultats expérimentaux.

3.6.1.7 `JackSat`

Approche innovante pour ce solveur qui ne fait pas une décomposition classique de l'arbre de recherche mais cherche à diviser le problème en sous-ensembles de variables pour former différents sous-problèmes [SM07]. Les sous-problèmes ayant moins de variables sont plus simples à résoudre. Chacun d'entre eux est calculé par un processeur puis les interprétations partielles trouvées sont jointes et vérifiées afin d'extraire, si elle existe, une solution globale. Une des difficultés consiste à générer des sous-ensembles de variables peu connexes. À noter qu'une version à mémoire partagée (avec OpenMP) est à l'étude.

3.6.2 Solveurs PRAM

Contrairement à la section précédente qui avait pour but de présenter quelques solveurs marquants, ici nous essayons de présenter une liste exhaustive des solveurs parallèles pour SAT dans un environnement multi-*threads*, à mémoire partagée. En 2008, les organisateurs de la SAT-Race ont dédié une partie de cette compétition pour les solveurs multi-*threads*. En 2009, les organisateurs de la SAT competition ont fait de même. Les résultats des sessions spéciales pour les solveurs parallèles de ces compétitions sont donnés ici.

3.6.2.1 `ySAT`

Premier solveur SAT programmé en mémoire partagée [FDH04], il fut écrit entièrement et non pas en reprenant une base de solveur existant. Toutefois, ses performances séquentielles étaient celles d'un solveur tel que `zchaff`, et donc très bonnes pour l'époque. Ce solveur maintient une liste des clauses apprises accessibles par l'ensemble des processeurs. Avec seulement quatre *threads* d'exécution, les processeurs perdent 10% de temps à attendre du travail. Ce travail est fourni par une liste de chemins de guidage que les processus vont récupérer depuis une liste mise à jour par les exécutions des autres *threads*. Les performances ne sont pas excellentes, mais l'erreur vient du fait que le solveur a été optimisé pour une exécution séquentielle. Les chercheurs ont

d'ailleurs fait le constat suivant : les optimisations de mémoire cache pour les solveurs séquentiels détériorent de façon drastique les performances des solveurs parallèles pour mémoire partagée. Solveur non présenté aux compétitions parallèles qui ont eu lieu en 2008 et 2009.

3.6.2.2 `MiraXT`

Ce solveur [LSB07] est la parallélisation du solveur `Mira` [LSB04, LSB05] et est orienté pour les formules industrielles. Un travail important a été réalisé afin d'offrir une base de clauses apprises et partagées accessible rapidement tout en minimisant les verrous nécessaires à sa gestion. Ce solveur utilise des structures paresseuses (*Watched Literal Reference List*) pour la mise à jour de la formule pendant la recherche de solution. Il a une efficacité d'environ 70% sur 2 processeurs et offre globalement de bonnes performances. Solveur présenté à la SAT-Race 2008 où il a terminé troisième (sur 3 solveurs présentés) mais offre des performances intéressantes. Il n'était pas représenté à la SAT competition 2009.

3.6.2.3 `PMinisat`

Parallélisation [CS08] simple de `Minisat 2.0` [SE08]. Les chemins de guidage sont conservés au sein d'une base partagée, elle même alimentée par l'exécution la plus longue dans l'arbre. Ce *thread* donne en priorité les sous-arbres les plus proches possibles de la racine. Les *threads* récupèrent un chemin de guidage depuis la base partagée lorsqu'ils en ont besoin. La particularité de ce solveur est dans la gestion de sa base de clauses : les clauses apprises par un processeur P_1 sont d'autant plus souvent partagées avec le processeur P_2 que ceux-ci partagent une grande partie du chemin de guidage. Cette idée provient du fait que certaines clauses ne sont utiles que localement dans un sous-arbre. Ce solveur a de meilleures performances que `MiraXT` et a terminé deuxième à la SAT-Race 2008. Non présent à la SAT competition 2009.

3.6.2.4 `ManySat`

Solveur [HJS08] basé sur le solveur `MiniSat 2.0` et par conséquent dédié à résoudre les formules industrielles, il procède à un échange des clauses apprises parmi les processeurs. L'échange se veut original : la politique de limitation des échanges est basée sur la taille, mais cette limite peut évoluer dans le temps en fonction de la qualité et du nombre d'échanges entre chaque processeur [HJS09b, HJS09a]. Ce solveur a aussi un comportement parallèle particulier comparé aux autres solveurs. Plutôt que de distribuer le travail issu de la décomposition de l'arbre de recherche, chaque processeur résout la totalité du problème mais avec un paramétrage différent pour chaque exécution. En effet, les performances d'un solveur moderne sont très sujettes à son paramétrage selon la formule à résoudre. Ce solveur a remporté les éditions SAT-Race 2008 et SAT competition 2009 des catégories pour solveurs industriels parallèles.

3.6.2.5 `satake`

Ce solveur [TOU09] arbore le classique principe maître/esclave. Les esclaves sont des exécutions classiques de `MiniSat 2.0` sur l'ensemble de la formule (donc pas de division de l'arbre), avec un espace privé des clauses apprises, et le maître s'occupe de la gestion des clauses partagées. La taille limite des clauses partagées est dynamique. De bonnes accélérations sont observées jusqu'à 8 processeurs, mais pas au-delà. De plus, il est impossible de lancer plus de 16 *threads* en parallèle. Ce solveur était présent à la SAT competition 2009 et avait de meilleures performances sur 4 processeurs que sur 16. Globalement, ce solveur ne rivalise pas avec `ManySat` alors qu'il en reprend les grands principes.

3.6.2.6 `ttsth`

Version parallèle [Spe09] du solveur `Ternary Tree Solver` [Spe08]. Ce solveur procède à une affectation statique des variables. L'ordre pré-établi correspond à un problème de minimisation d'hypergraphe, mais ce problème est plus difficile à résoudre que SAT lui-même, alors l'auteur a recours à une heuristique pour cette partie. Une fois que l'ordre est établi, l'algorithme procède à la construction d'un arbre ternaire dont les nœuds sont les variables et les branches sont des ensembles de clauses selon qu'elles contiennent le littéral positif (branche gauche) ou négatif (branche droite) de la variable associée au nœud. Les clauses ne contenant pas la variable sont dans la branche du milieu. Cette construction est faite à l'aide de tables de hachage. L'exploration d'un tel arbre permet de savoir si la formule admet une solution ou pas. Seule cette partie d'exploration est traitée en parallèle dans la version idoine, mais le mécanisme n'est pas expliqué en profondeur. Sans plus de détails, il semblerait que le principe soit basé sur un parcours simultané de l'arbre ternaire par plusieurs *threads*. Présent à la SAT competition 2009, ce solveur offre de piètres performances comparé à `ManySat` (moins de 3% des formules testées sont résolues alors que `ManySat` en résout plus de 60% sur le même ensemble de formules [1].

3.6.2.7 `gNovelty+-T`

Solveur [PG09] très récent ayant la particularité d'être incomplet et destiné à résoudre les formules générées aléatoirement. Le solveur `gNovelty+` [PTGS08] sur lequel celui-ci est basé n'explore pas un arbre de recherche puisqu'il effectue une recherche locale stochastique. Il utilise un mécanisme de poids sur les clauses non satisfaites, incrémenté à chaque fois que la clause a été testée non satisfaite. Cela améliore le choix de la prochaine variable dont la valeur sera inversée. Le parallélisme de ce solveur est simplement l'exécution concurrentes et indépendantes de plusieurs instances de ce solveur, certainement avec des graines aléatoires différentes. Ce solveur était le seul présent pour la catégorie des formules aléatoires de la SAT competition 2009.

1. `http://www.cril.univ-artois.fr/SAT09/results/globalbysolver.php?idev=23&det=2`

3.6.3 Synthèse

Afin de mieux juger des spécificités de chaque solveur, voici les quelques solveurs sus-cités représentés sous forme de tableau (voir tableau 3.1). La première colonne donne le nom du solveur, la seconde l'année de publication, la colonne « Prog. » indique si le solveur est prévu pour fonctionner sur des machines à mémoire distribuée (DRAM) ou à mémoire partagée (PRAM). La colonne « Architecture » permet de spécifier si le solveur a une spécificité qui le réserve à un type de machines ou de topologie de réseau. La colonne « Complet ? » indique si oui ou non le solveur est complet (il prouve l'insatisfaisabilité s'il est complet). La colonne « Application » montre pour quel type de formule le solveur est dédié ; si aucune colonne ne montre les solveurs capables d'échanger des clauses, c'est parce que tous les solveurs industriels le font. La colonne « Décomposition » précise la manière dont l'algorithme dégage du parallélisme pour ses processeurs. La colonne « Parallélisme » indique si l'algorithme adopte une stratégie collaborative ou concurrentielle et la colonne « Équilibrage » décrit la technique employée pour équilibrer le travail entre les processeurs.

TABLE 3.1 – Solveurs SAT parallèles et leurs spécificités

Nom	Année	Prog.	Architecture	Complet ?	Application	Décomposition	Parallélisme	Équilibrage
Böhm, Speckenmeyer	1994	DRAM	NoRMA	Oui	Aléatoires	Sous-arbres	Coopératif	Serveur initiative : estimation
PSATO	1994	DRAM	NoRMA	Oui	Aléatoires	Sous-arbres	Coopératif	Maître / esclave : chemin de guidage
//satz	2001	DRAM	NoRMA	Oui	Aléatoires	Sous-arbres	Coopératif	Maître / esclave : chemin de guidage
PaSAT	2001	DRAM	NoRMA	Oui	Industrielles	Sous-arbres	Coopératif	Maître / esclave : chemin de guidage
GridSat	2003	DRAM	Grille	Oui	Industrielles	Sous-arbres	Coopératif	Maître / esclave : chemin de guidage
PSolver	1998	DRAM	Pair à pair	Oui	Selon client	Sous-arbres	Coopératif	Maître / esclave : chemin de guidage
JackSat	2007	DRAM	NoRMA	Oui	Industrielles	Variables	Coopératif	Centralisé : statique
ySAT	2004	PRAM	CC-(N)UMA	Oui	Industrielles	Sous-arbres	Coopératif	Serveur initiative : chemin de guidage
MiraXT	2007	PRAM	CC-(N)UMA	Oui	Industrielles	Sous-arbres	Coopératif	Source initiative : chemins de guidage
PMinisat	2008	PRAM	CC-(N)UMA	Oui	Industrielles	Sous-arbres	Coopératif	Serveur initiative : chemin de guidage
ManySat	2008	PRAM	CC-(N)UMA	Oui	Industrielles	Non	Concurrentiel	Non applicable
satake	2009	PRAM	CC-(N)UMA	Oui	Industrielles	Non	Concurrentiel	Non applicable
ttsth	2009	PRAM	CC-(N)UMA	Oui	Industrielles	Chemins	Coopératif	Serveur initiative : arbre ternaire partagé
gNovelty+-T	2009	PRAM	CC-(N)UMA	Non	Aléatoires	Non	Concurrentiel	Non applicable

3.7 Conclusion

SAT est un domaine très concurrentiel, très étudié dans un cadre séquentiel, et plus tardivement étudié dans un cadre parallèle. Comme montré dans ce chapitre, mettre en œuvre un algorithme parallèle pour SAT permet d'obtenir des gains super linéaires. Cela signifie que le parallélisme n'est pas qu'une simple solution technique pour résoudre plus rapidement le problème SAT mais qu'il apporte un réel gain algorithmique à sa résolution. La découverte d'une solution pour les formules satisfaisables donne parfois des gains super linéaires tandis que dans le cas de formules insatisfaisables, dans certains cas de figures de tels gains sont également envisageables. Peut-être que les chercheurs intéressés par SAT n'ont globalement pas la culture du développement d'algorithmes parallèles mais avec l'arrivée des processeurs multi-cœurs dans l'informatique grand public aujourd'hui, cela deviendra naturel. Cette thèse s'inscrit donc dans le double but de tirer le maximum de performances des processeurs multi-cœurs mais aussi d'accélérer au maximum la résolution de SAT. La recherche pour les solveurs SAT orientés multi-cœurs (ou pour mémoire partagée en général) en est à ses balbutiements. La quasi totalité des solveurs actuels sont dédiés à résoudre des formules industrielles, seulement un seul résout des formules générées aléatoirement, mais de manière incomplète. Il n'existe pas aujourd'hui à notre connaissance de solveur pour machine multi-cœurs à mémoire partagée complet pour résoudre les formules aléatoires. De plus, la manière dont la majorité de ces solveurs fonctionne peut surprendre : plutôt que de coopérer à plusieurs processeurs pour trouver une solution plus rapidement, les processeurs sont concurrents et travaillent sur différentes copies de la même formule. Cette approche a un sens pour des algorithmes incomplets ou pour des solveurs industriels car dans un cas, il y a beaucoup d'aléatoire dans le processus de recherche, et dans l'autre, le paramétrage du solveur peut donner des écarts de performance très imprévisibles en fonction de la formule à résoudre. Par conséquent, oui cette approche a un sens pour ces types de solveur, mais un solveur de type complet pour résoudre des formules aléatoires se doit d'être collaboratif pour exploiter le parallélisme offert par le multi-cœurs. C'est exactement ce type de solveur que nous avons élaboré au cours de cette thèse, et ce sont maintenant les grands principes qui ont guidé le développement de MTSS qui seront discutés au chapitre suivant.

Chapitre 4

MTSS : Multi-Threaded SAT Solver, principes généraux

4.1 Introduction

L'efficacité des algorithmes pour résoudre SAT n'a pas cessé d'augmenter depuis la naissance de cet axe de recherche. Les gains sont imputables aux nouveaux algorithmes, mais aussi à la croissance constante de la puissance brute des microprocesseurs. Comme nous l'avons énoncé dans le chapitre 2, l'avenir des micro-processeurs n'est plus dans l'augmentation de leur vitesse de traitement mais dans l'augmentation du nombre de cœurs de calcul. Tant que les micro-processeurs ne subiront pas de changement de technologie, les algorithmes séquentiels actuels seront limités en performance, ils ne seront plus accélérés par les machines, ou très peu. Il est donc temps d'imaginer de nouveaux algorithmes dont les performances seraient proportionnelles au nombre de cœurs utilisés. Dans le même temps, il faut garder à l'esprit que la majorité des chercheurs dans le domaine SAT ont une bonne connaissance de la programmation séquentielle. Il est par conséquent nécessaire que les améliorations apportées par la communauté dans les algorithmes séquentiels puissent être implémentables dans ce nouvel algorithme. C'est dans cette optique que nous avons développé le solveur MTSS (pour *Multi-Threaded* SAT *Solver*). C'est un algorithme capable de tirer profit des cœurs disponibles dans un ordinateur. Dans le même temps, il reste évolutif afin de faciliter l'intégration de nouvelles fonctions de manière simple et rapide. L'explication pratique de la mise en œuvre d'un tel algorithme sera détaillée dans le chapitre 5. Tout d'abord, nous devons discuter des principes que nous avons définis afin que MTSS remplisse son rôle tel qu'il vient d'être décrit. Nous introduisons le modèle riche / pauvre qui permet de multiplier les traitements possibles dans l'arbre de recherche et la notion d'arbre de guidage qui donne à ce modèle les moyens de développer en parallèle un arbre binaire de recherche. Ces deux concepts ont déjà fait l'objet de publications, ils ont été introduits dans [VSDK08] et approfondis dans [VSDK09a].

4.2 Double objectifs

4.2.1 Démocratisation du multi-cœurs

Cette thèse s'inscrit dans l'optique d'utiliser au mieux les processeurs multi-cœurs. Comme expliqué en section 2.2, ce type de processeurs devient la norme aujourd'hui, mais ce n'est pas une raison suffisante. Il a été montré tout au long du chapitre 3 que le parallélisme apporte un réel gain dans la résolution de SAT. En réalité, ce n'est pas par dépit qu'il est intéressant de programmer SAT en parallèle, mais c'est plutôt une opportunité à saisir. En effet, les architectures du type CC-UMA offrent d'excellentes performances et permettent la création d'algorithmes non implémentables auparavant, mais tout cela à un coût en général élevé. Avec l'arrivée des processeurs multi-cœurs, de telles architectures deviennent abordables et se démocratisent. De plus, ce sont les briques les plus petites des récents super calculateurs type NoRMA ou CC-NUMA. Les avantages sont simples : plusieurs unités de calcul et surtout une même mémoire rapide d'accès. Dans de telles conditions, les chercheurs ne sont plus limités par les messages. Le nombre et la taille des messages ne sont pas un dilemme. Le solveur MTSS (*Multi-Threaded* SAT *Solver*) est conçu pour profiter des avantages offerts par ces nouveaux micro-processeurs.

4.2.2 Performances séquentielles

Le chapitre 1 avait pour but d'expliquer le problème SAT, nous avons aussi présenté quelques concepts efficaces qui aident à la résolution de SAT parmi la multitude de recherches existantes dans ce domaine. C'est pourquoi nous nous sommes fixés un second objectif en sus d'utiliser au mieux les possibilités des processeurs multi-cœurs : celui de pouvoir réutiliser facilement tout ce qui a déjà été imaginé dans le cadre séquentiel dans un cadre parallèle. MTSS est donc conçu dans le but de proposer un cadre de parallélisation simple à appréhender. Il prend en charge la parallélisation du problème mais la manière de le résoudre est laissée à l'implémentation.

4.3 Algorithme fortement collaboratif de MTSS

4.3.1 Principe général

Les architectures CC-UMA permettent de mettre en place un solveur SAT très fortement collaboratif. Nous voulons travailler sur un algorithme complet pour résoudre les formules générées aléatoirement. Dans le même temps, nous souhaitons aussi pouvoir contribuer à la communauté en conservant un certain nombre de concepts connus afin que la communauté puisse s'approprier en retour ce nouvel algorithme. Nous avons donc fait le choix naturel de conserver une exécution séquentielle classique d'un solveur complet. Autour de ce solveur, nous mettons en place différents traitements dont le but est de faciliter l'exploration de l'arbre binaire de recherche pour ce solveur. Les traitements n'étant pas clairement identifiés, cela permet une grande souplesse dans

l'approche que l'on souhaite donner à MTSS. Il en résulte une description plus formelle, que voici.

Définition 4.1 *Le **processus riche** est capable de conclure sur la nature logique de la formule étudiée dans un temps fini mais d'ordre exponentiel.*

Exemple 31 *Un solveur du type* DLL *est un processus riche.*

Définition 4.2 *Un **processus pauvre** fournit une information partielle ou définitive valable sur tout ou partie de l'espace de recherche.*

Exemple 32 *Une propagation, un look-ahead, le choix d'une variable de branchement, l'apprentissage d'une clause ou un solveur type* `Walksat` *sont autant d'exemples de processus pauvres.*

Les termes de « riche » et « pauvre » sont choisis par rapport à la qualité de l'information qu'ils ont pendant la résolution. En effet, le processus riche, que nous appellerons dorénavant le riche connait la valeur logique courante de la formule, il suit une recherche ordonnée (parcours main gauche) dans l'arbre de recherche. Les processus pauvres (que nous nommerons pauvres) n'ont qu'une vue partielle de la formule puisqu'ils travaillent localement à un endroit dans l'arbre pour ensuite être affectés à une autre tâche à un autre endroit dans l'arbre. La seule règle à laquelle ils obéissent est de travailler en avance du riche afin de l'aider dans sa recherche de solution. Comme présenté ici, MTSS a un profil très collaboratif dans un environnement où le grain est extrêmement fin, il est de l'ordre d'un traitement local. Nous ne souhaitons pas paralléliser les fonctions utilisées dans les solveurs SAT car si une amélioration est apportée au modèle séquentiel, alors il faudrait retravailler sur la version parallèle. Or, en considérant les traitements locaux comme des éléments non subdivisibles en parallèle, mais seuls leurs agencements et leurs exécutions pouvant être parallélisées, alors il est très simple de travailler et d'améliorer ces traitements. MTSS dresse un cadre générique pour solveur SAT parallèle.

4.3.2 Le processus riche

Le riche poursuit un processus de résolution systématique. Dans le cas de MTSS, nous avons implémenté une procédure du type DLL. Son comportement assure la complétude de l'algorithme MTSS puisqu'il explore la totalité de l'espace de recherche (mais il peut être aidé par d'autres processus pour cela). Tout comme un DLL, le processus riche de MTSS fera quelques traitements locaux, choisira des variables de branchement afin d'approfondir la recherche de solution et effectuera des *backtracks* lorsque le sous-arbre en cours de calcul contient une clause vide. La seule différence avec un DLL classique se situe au niveau de la gestion du *backtrack* : en lieu et place d'un simple *backtrack* pour changer la valeur d'une précédente variable de branchement, le riche vérifie au préalable si un pauvre a laissé de l'information concernant le sous-arbre qu'il s'apprête à calculer. L'échange d'information dans notre approche très fine de collaboration s'opère, pour le riche, à cet instant clé (voir algorithme 6). Trois cas sont à distinguer :

- Aucune information calculée : le riche continue alors son exploration de l'arbre de recherche dans l'autre branche de ce niveau (cas normal du *backtrack* d'un algorithme basé sur DLL).
- Un pauvre est en cours de calcul sur le nœud que doit reprendre le riche : le riche donne son rôle au pauvre rencontré et devient lui-même un pauvre (détails techniques en section 5.4.3) si le calcul du pauvre correspond à ce que le riche aurait dû faire.
- Information disponible (calculée par un ou plusieurs pauvres) : cette information peut être seulement le contexte de calcul après simplification plus l'identité de la prochaine variable de branchement ou la valeur logique du sous-arbre ; ou encore n'importe quel état intermédiaire entre ces deux extrêmes. Voir la section 5.4.1 pour plus de détails techniques.

Dans le dernier cas, étiqueté « Échange de Contexte » dans l'algorithme 6, le riche utilise les informations laissées par les calculs des pauvres dans un contexte de calcul. Le riche reprend ensuite son calcul à partir du nœud auquel le contexte fait référence. Afin de ne pas stopper le processus riche à cause d'un pauvre qui n'aurait pas terminé une tâche au niveau d'un nœud, le riche procède à l'échange de rôle. Voir la partie de l'algorithme 6 étiquetée par « Échange de Rôle ».

Le procédé fortement collaboratif mis en place entre les deux types de processus permet de minimiser les blocages entre le riche et les pauvres. Cette vision permet aussi de garder un aspect séquentiel au solveur ; comme énoncé dans les objectifs de MTSS, il nous paraissait important de pouvoir étendre les possibilités de MTSS par la communauté SAT, sans que cela demande de grandes compétences en algorithmique parallèle. En effet, cette partie du solveur se veut très peu sujette aux surcoûts parallèles. Dans notre vision, le riche ne doit pas perdre de temps à donner de l'information aux pauvres mais doit pouvoir gagner un maximum de temps grâce à eux.

4.3.3 Les processus pauvres

Le pauvre fournit une information partielle ou définitive sur tout ou partie de l'espace de recherche (par exemple : propagation unitaire, heuristique de branchement, apprentissage de clauses, traitement local, ...). C'est un processus volatile qui explore l'arbre de recherche pour trouver un nœud pendant, pas encore calculé pour y calculer une tâche présentant un grain fin ou même un gros grain de parallélisme. Il est capable d'appliquer un traitement particulier au niveau de ce nœud ou de calculer plusieurs nœuds jusqu'à un sous-arbre complet. Ce traitement n'est pas défini, toutes les améliorations à la résolution de SAT sont envisageables ici. Que ce soit un traitement incomplet local ou global, un traitement complet local, il n'y a pas de limite aux capacités des pauvres et pas de limite non plus au grain des tâches qu'ils peuvent effectuer.

Les pauvres n'ont pas de comportement séquentiel à proprement parlé, un tel processus n'a pas de sens dans un contexte séquentiel car l'ordre des tâches qu'il effectue n'est pas déterminé avant l'exécution de l'algorithme. Les pauvres travaillent en commun pour pré-traiter le maximum d'information et la résumer afin que le riche puisse être accéléré au maximum. L'approche mise en place pour MTSS est parallèle grâce

Algorithme 6 Processus Riche

Entrée : $\mathcal{F}$: une formule CNF
PROCESSUSRICHE($\mathcal{F}$)
si $\mathcal{F}$ contient un littéral monotone l **alors**
retourner PROCESSUSRICHE($\mathcal{F}\backslash l$) **(Monotonie)**
sinon si $\mathcal{F}$ contient une clause unitaire contenant l **alors**
retourner PROCESSUSRICHE($\mathcal{F}\backslash l$) **(Propagation Unitaire)**
sinon si $\mathcal{F}$ contient au moins une clause vide **alors**
retourner FAUX**(Backtrack)**
sinon si $\mathcal{F}$ est vide **alors**
retourner VRAI**(Solution)**
sinon
$v \leftarrow$ une variable non affectée de $\mathcal{F}$ **(Choix)**
si PROCESSUSRICHE($\mathcal{F}\backslash v$) = VRAI **alors**
retourner VRAI
sinon si Au moins un pauvre a terminé son calcul sur le nœud courant **alors**
Remplacer le contexte de calcul par celui du pauvre **(Échange de Contexte)**
sinon si Le nœud courant est en cours d'un calcul équivalent au riche par un pauvre **alors**
Indiquer un échange de rôle sur le nœud
Devenir un PROCESSUS PAUVRE **(Échange de Rôle)**
sinon
retourner PROCESSUSRICHE($\mathcal{F}\backslash \bar{v}$)
fin si
fin si

au concept des processus pauvres, d'ailleurs le nombre de pauvres n'est théoriquement pas limité. Nous avons vu que la synchronisation du riche est des pauvres se faisait lors du *backtrack* du riche. Les pauvres entre eux n'ont pas besoin de se synchroniser explicitement. Ils laissent l'information qu'ils ont calculée, ensuite elle sera utilisée par un pauvre ou par le riche, peu importe. Il n'y a qu'une gestion de verrous afin d'éviter que plusieurs processus pauvres ne travaillent sur le même nœud au même instant. Afin que les pauvres puissent travailler en assez grand nombre en parallèle, il a été introduit le concept d'*arbre de guidage* qui sera détaillé en section 4.4.2. Vous rencontrerez cette notion dans l'algorithme du processus pauvre donné par l'algorithme 7.

Algorithme 7 Processus Pauvre

Entrée : $\mathcal{F}$: une formule CNF
Entrée : T, une tâche
  PROCESSUSPAUVRE($\mathcal{F}$, **T**)
  $n \leftarrow$ racine de l'arbre de recherche de $\mathcal{F}$
  tantque $\mathcal{F}$ ne possède pas de solution **faire**
    si T peut être appliquée sur n **alors**
      Appliquer T sur n
      si Le Riche a indiqué un échange de rôle sur n **alors**
        Devenir le PROCESSUS RICHE **(Échange de Rôle)**
      fin si
    fin si
    si n est le nœud en cours de calcul par le RICHE **alors**
      $n \leftarrow$ racine de l'arbre de recherche de $\mathcal{F}$
    sinon
      Étendre le *sous-arbre de guidage* enraciné en n **(Extension de l'arbre de guidage)**
      $n \leftarrow$ prochain nœud du *chemin de guidage*
    fin si
  fin tantque

4.3.3.1 Conclusion

Le schéma de fonctionnement décrit ci-dessus propose un cadre générique de solveur SAT parallèle. Le processus riche peut être choisi parmi les solveurs SAT complets actuels, tandis que les processus pauvres peuvent être tout traitement local ou encore tout ou partie d'un solveur existant. Bien entendu, il n'est pas interdit d'intégrer de nouveaux traitements ou solveurs au sein de cette approche. Le cadre étant mis en place, il est maintenant nécessaire de gérer la distribution mais surtout l'équilibrage de charge entre ces processus. Il est évident que le grain parallèle sera très fin puisque les processus pauvres sont censés exécuter des fonctions composées de relativement peu d'instructions. Ces observations nous ont amené à devoir imaginer un nouvel objet parallèle expliqué dès maintenant.

4.4 Un nouvel objet parallèle

Les solveurs SAT parallèles sont confrontés à un défi de taille, celui de donner assez de travail à toutes les unités de calcul présentes dans l'ordinateur sur lequel il est exécuté. L'objet le plus utilisé de nos jours servant à distribuer le travail est le chemin de guidage. Cet objet a été décrit plus en détails en section 3.4.2. À partir d'un chemin, il est possible de récupérer les chemins désignant les sous-arbres non développés pour ensuite les affecter à des processeurs oisifs. Cette structure présente néanmoins quelques limites, en particulier dans le cadre du solveur MTSS tel que nous l'avons défini.

4.4.1 Inconvénients du chemin de guidage

Voici les deux principaux inconvénients :

- Les espaces de recherche qui n'ont pas encore été explorés (représentés par les sous-arbres pendants dans le chemin de guidage) sont de tailles différentes. En effet, le sous-arbre de droite immédiatement sous la racine sera de taille très importante alors que les espaces de recherche proches des feuilles seront très petits. Cela conduit inévitablement à un grand déséquilibre entre les tâches données aux unités de calcul, et provoque une perte d'efficacité parallèle . Plus formellement, si une instance SAT contient n variables, à la profondeur i dans l'arbre, il y a $2^{(n-i)}$ instanciations possibles à tester. Avec un exemple de seulement 10 variables, il existe au total $2^{10} = 1024$ instanciations à tester à partir de la racine alors qu'il n'y en a plus que $2^7 = 128$ après seulement 3 variables fixées (il y a un facteur $2^3 = 8$ instanciations de moins à vérifier après 3 étapes).
- Une des conséquences des heuristiques de branchement efficaces est la réduction en hauteur des arbres de recherche. Il en résulte pour le chemin de guidage un plus petit nombre de sous-arbres pendants à développer. C'est-à-dire que le nombre de tâches parallèles est réduit.

Pour résoudre le premier point, les solveurs font de l'équilibrage dynamique de la charge afin de subdiviser les sous-arbres de grandes tailles. Mais le deuxième point est un problème, même pour les solveurs faisant un bon équilibrage de charge. Le processus maître est obligé de stopper souvent les premières exécutions du solveur pour avoir suffisamment de chemins de guidage afin d'alimenter un nombre important d'unités de calcul. Pour résumer, le chemin de guidage est intrinsèquement un objet séquentiel puisque, dans un chemin, chaque nœud dépend de son prédécesseur. Pour équilibrer la charge, les solveurs sont contraints de développer des techniques destinées à couper les chemins de guidage, soit pour augmenter le nombre de tâches, soit pour réduire le travail d'un processeur.

Le chemin de guidage utilisé dans le contexte des définitions de processus riche et pauvres conduit aux problèmes donnés ci-dessus (voir la figure 4.1) puisque le seul réel chemin qui sera déployé pendant la recherche de solution par MTSS sera celui du processus riche.

- Les sous-arbres de tailles différentes généreront des tâches très déséquilibrées pour les processus pauvres. Certains seront oisifs alors que d'autres auront beaucoup

de travail.
– La longueur du chemin du riche empêchera de faire travailler un grand nombre de processus pauvres simultanément.

Les problèmes sont donc déséquilibre et famine. De plus, et c'est le principal inconvénient du chemin de guidage pour notre approche fortement collaborative : les processus pauvres n'ont pas vocation à travailler sur l'ensemble d'un sous-arbre. Le grain de parallélisme de notre approche est au niveau d'un traitement local. Le principe du chemin de guidage stipule que le processus qui souhaite travailler dans l'arbre doit partir de la formule de départ puis propager les variables du chemin de guidage qu'il reçoit. Ces propagations supplémentaires pour préparer le travail d'un pauvre multipliées par le nombre de tâches effectuées par les processus pauvres ferait exploser le nombre de propagations dans le processus de résolution. Il en résulterait un effondrement des performances.

Pour l'approche riche / pauvre donnée ici, nous avons besoin d'une autre méthode pour équilibrer la charge de travail dynamiquement sur un grand nombre de cœurs de calcul. De plus, les systèmes multi-cœurs ne sont pas encore constitués de centaines d'unités de calcul, nous ne souhaitons donc pas mettre en place une procédure maître / esclave qui monopoliserait un processeur dédié à l'équilibrage de charge. Se passer d'un cœur de calcul dans la résolution de la formule est très pénalisant. Comme expliqué plus haut, nous souhaitons que le processus riche soit le moins perturbé par le surcoût du parallélisme, il ne faut donc pas qu'il soit sollicité pour fournir des tâches aux pauvres. La méthode d'équilibrage de charge dynamique dans le contexte riche / pauvre sera de type serveur initiative : les pauvres souhaitant travailler iront chercher eux-mêmes les nœuds disponibles pour le travail.

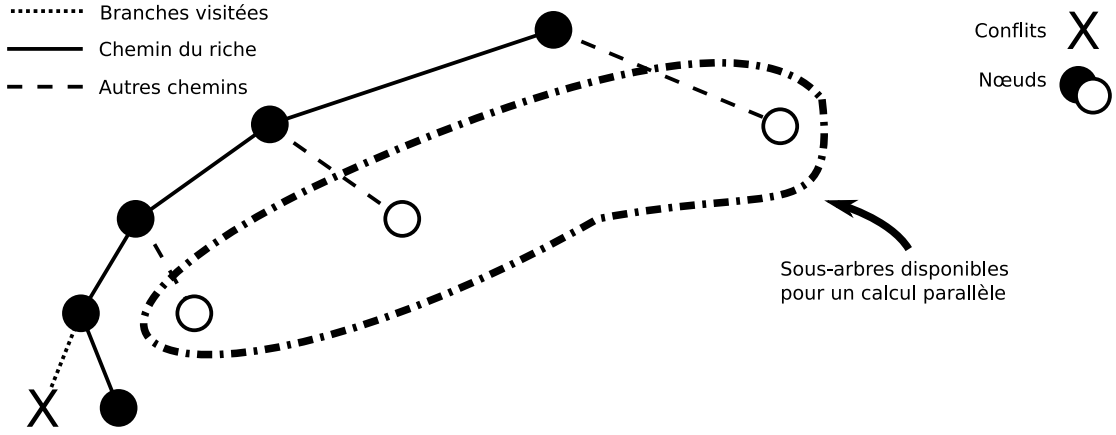


FIGURE 4.1 – Exemple d'un chemin de guidage

4.4.2 L'arbre de guidage

Nous présentons un nouveau moyen de distribuer le travail parmi plusieurs cœurs de calcul. Le nom de cette méthode est l'arbre de guidage. En effet, ici, au lieu de partager

un chemin afin de travailler sur celui-ci, à l'instar du chemin de guidage, l'arbre en cours de développement est totalement partagé entre les processus dans notre méthode, il fait donc office d'arbre de guidage. Les processus pauvres se guident sur l'arbre afin de découvrir des nœuds pendants sur lesquels ils peuvent effectuer une tâche.

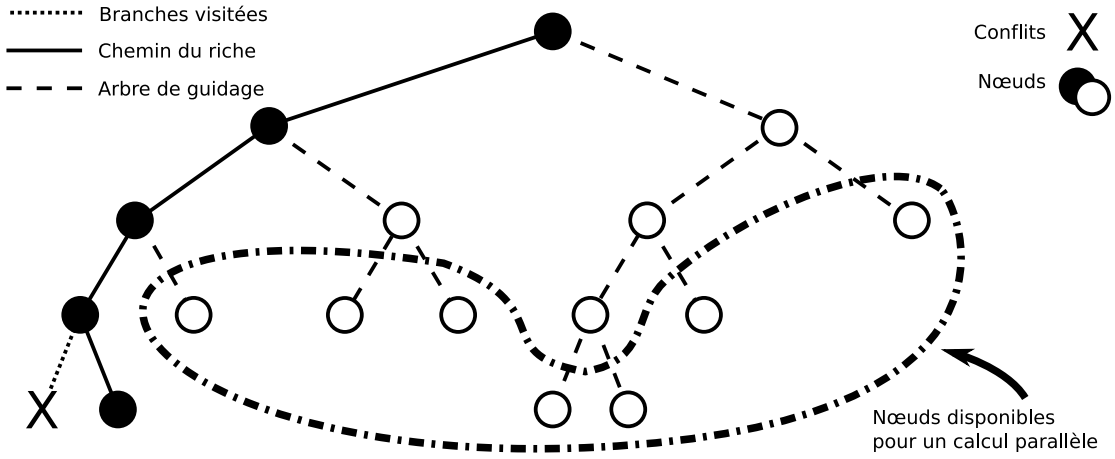


FIGURE 4.2 – Exemple d'un arbre de guidage

L'arbre de guidage est un objet parallèle spécialement conçu pour les machines multi-cœurs à mémoire partagée car les données doivent être toujours complètement connues par l'ensemble des unités de calcul. Cet objet n'est pas naturellement implémentable dans un modèle à mémoire distribuée puisque le nombre de messages échangés serait astronomique pour maintenir son état parmi les processeurs. L'arbre de guidage est un arbre de recherche développé simultanément par plusieurs entités (figure 4.2). Chacune d'entre elles calcule un nœud : simplification de la formule par la variable de branchement à l'une de ses deux valeurs suivie du choix de la prochaine variable de branchement. L'arbre de guidage contient le chemin de guidage comme le squelette de sa structure, mais plutôt que d'avoir plusieurs sous-arbres, chacun d'eux affecté à un processeur, ici les sous-arbres sont calculés par plusieurs processeurs simultanément. En réalité, chaque processeur ne calcule qu'un nœud à la fois puis peut en calculer un autre dans un autre sous-arbre. Ce procédé permet de dégager plus de parallélisme, dans le sens où le nombre de tâches exécutables en parallèle est plus important que dans la définition du chemin de guidage. En utilisant un arbre de guidage, il n'est pas nécessaire de maintenir une liste de tâches à distribuer. Il suffit aux pauvres d'explorer l'arbre de guidage pour les trouver. Le nombre de tâches disponibles à un instant t est égal au nombre de nœuds dans l'arbre additionné de un moins le nombre de sous-arbres déclarés insatisfaisables à cet instant.

Idée de la preuve: Soient n_i le nombre de nœuds d'un arbre et b_i le nombre de branches pendantes dans ce même arbre après l'ajout de i nœuds dans cet arbre. Trivialement, à la racine ($i = 1$) on a $n_1 = 1$ et $b_1 = 2$. Par construction, si un nœud est ajouté, cela signifie qu'un nœud est venu se greffer au bout d'une branche pendante, à chaque étape une seule branche pendante est donc remplacée par deux branches

pendantes, cela revient à n'ajouter qu'une branche pendante. On pose donc les suites mathématiques suivantes : $n_{i+1} = n_i + 1$ et $b_{i+1} = b_i + 1$. Posons $b_1 = n_1 + 1$, on obtient alors $b_{i+1} = n_i + 1 + 1$ et par une nouvelle substitution, on obtient $b_{i+1} = n_{i+1} + 1$. On conclue donc que $\forall i \in \mathbb{N}$, le nombre de branches pendantes est égal au nombre de nœuds plus un dans un arbre binaire. Cette relation est vraie même pour $i = 0$ si on considère que la première branche pendante est la formule elle-même.

Problème : ce nombre de branches pendantes comptabilise aussi les branches déclarées insatisfaisables. Or ces branches ne sont plus des tâches à résoudre pour les unités de calcul. On a donc bien le nombre de tâches disponibles à un instant donné égal au nombre de nœuds dans l'arbre additionné de un moins le nombre de sous-arbres déclarés insatisfaisables à cet instant.

□

L'arbre de guidage précédemment décrit peut-être divisé en deux éléments distincts : le chemin du riche et les sous-arbres qui y sont enracinés. Le chemin du riche, conformément à sa définition, est développé de la même manière que le ferait un solveur SAT séquentiel classique (inspiré de DLL). Les sous-arbres enracinés dans le chemin de guidage, formant ensemble l'arbre de guidage, sont déployés simultanément par les processus pauvres. L'arbre de guidage résulte du travail fortement collaboratif de l'ensemble des processus de MTSS. Cet objet permet en outre aux pauvres de pouvoir calculer des tâches de grains différents. Leur caractéristique de pouvoir être n'importe quel traitement séquentiel existant est donc possible dans le contexte de l'arbre de guidage.

Encore une fois, afin de ne pas freiner le riche dans sa recherche, les pauvres adoptent le principe du chemin de guidage sur le chemin du riche, à savoir qu'ils recalculent la formule locale à partir de la formule d'origine simplifiée par le chemin du riche. Ensuite, les résultats des calculs des pauvres sont déposés dans l'arbre de guidage par le dépôt d'un contexte de calcul au niveau du nœud qu'ils ont calculé. De cette manière, il n'y a pas recalcul à chaque fois que les pauvres ont une tâche à réaliser dans un sous-arbre de guidage.

Alors que le chemin de guidage est un objet développé séquentiellement afin de paralléliser le calcul des sous-arbres, l'arbre de guidage n'a pas de sens en séquentiel et est immédiatement un objet parallèle. L'équilibrage de charge n'est pas explicite, il est naturel. Le riche aura toujours du travail puisqu'il suit un ordre prédéfini dans l'arbre (parcours main gauche, comme tous les solveurs basés sur DLL). Les pauvres parcourent l'arbre de guidage à la recherche d'un nœud pendant. Chaque sous-arbre de guidage contient forcément un nœud pendant et donc une tâche à calculer car dans le cas contraire, toutes les feuilles ont conduit à des conflits et l'information sera remontée jusqu'au niveau du nœud appartenant au chemin de guidage. En effet, le développement de l'arbre est fait par les processus pauvres, mais le développement concerne aussi les *backtracks*.

4.5 Conclusion

Les ordinateurs à mémoire partagée ayant la particularité d'offrir à tous les processeurs un accès rapide aux mêmes objets durant l'éxecution d'un programme, les algorithmes dédiés ne sont pas limités par le nombre de messages échangés des ordinateurs à mémoire distribuée. Il est donc possible d'imaginer des algorithmes pleinement collaboratifs travaillant ensemble sur un même objet de grande taille. C'est ce que nous proposons avec l'arbre de guidage qui est un objet totalement parallèle, mais où le travail par nœud est séquentiel. Cet objet est mis à jour par deux types de processus : le riche qui garantit la complétude et la terminaison de l'algorithme et les pauvres qui aident à la résolution du problème. Le schéma riche / pauvre présenté ici est à différencier du classique schéma maître / esclave car le riche n'est pas simplement destiné à fournir du travail et à gérer les processus pauvres, il résout la formule. D'ailleurs, ce principe riche / pauvre en version séquentielle se résume à exécuter le processus riche. Les pauvres ne sont pas de simples esclaves car ils n'effectuent pas tous la même tâche donnée par un processus maître, la définition est plus large, une multitude de traitements est envisageable. Chaque pauvre exécute n'importe quel type de traitement dans l'arbre, sans connaître l'intégralité du problème. Les tâches peuvent être de n'importe quelle taille de grain parallèle : du traitement local d'un nœud au calcul d'un sous-arbre. Le processus pauvre se charge lui-même de trouver du travail dans l'arbre, le travail ne lui est pas explicitement demandé par le riche. La diversité des implémentations possibles est intéressante et est réalisable grâce aux deux principes discutés ici, à savoir l'approche riche / pauvre et l'arbre de guidage. Il est maintenant intéressant de savoir comment de tels mécanismes et rouages ont pu être mis en place dans un solveur réellement programmé et quels ont été les défis à relever. Nous allons maintenant décrire les particularités du solveur que nous avons développé dans ce cadre général de solveur SAT parallèle.

Chapitre 5

MTSS, notre mise en œuvre

5.1 Introduction

Nous avons vu que les nouveautés apportées par MTSS sont la différenciation de deux types de processus (riche et pauvres) et leur collaboration pour explorer l'arbre de recherche qui crée l'arbre de guidage, ainsi que la possibilité d'ajouter des tâches spécifiques aux pauvres. Cela donne au cadre que nous avons mis en place beaucoup de souplesse, de nombreuses implémentations différentes reprenant ces principes sont possibles. Ce chapitre décrit les choix que nous avons faits pour notre implémentation de MTSS. Nous décrivons la manière dont l'arbre de guidage a été implémenté (section 5.3.2) afin de fournir un objet performant aux processus de MTSS. La synchronisation mise en place entre les tâches pour minimiser les attentes entre processus malgré une très forte collaboration est présentée en section 5.4. Les tâches pauvres intégrées actuellement à MTSS sont exhaustivement répertoriées en section 5.5. Parmi les tâches disponibles, nous pouvons différencier celles présentant un grain de parallélisme très fin qui construisent l'arbre de guidage, et une autre à gros grain permettant de paralléliser simplement un solveur SAT externe à MTSS. Tout d'abord, nous allons présenter les caractéristiques du cœur de MTSS, le processus riche en tant que solveur DLL classique. La majorité de ces informations d'implémentation ont été publiées, et plus particulièrement, l'outil de parallélisation de solveur externe le fût dans [VSDK09b, VSDK09c].

5.2 Caractéristiques du solveur DLL

5.2.1 Heuristique de branchement

Comme nous le verrons plus tard, MTSS est capable d'utiliser toutes sortes d'heuristiques (section 5.5.5), mais celle implémentée par défaut dans MTSS est BSH [DD06]. La version implémentée ne comprend pas toutes les améliorations et optimisations que possède `kcnfs` [DD03] (le solveur intégrant le BSH d'origine), mais le principe est le même. MTSS sélectionne un groupe de variables (peut-être l'ensemble des variables, par exemple à la racine) pour lesquelles on souhaite connaître le nombre de clauses réduites ainsi que leur taille une fois réduite. À partir de ces informations un calcul est appliqué

pour sélectionner celle qui aura le plus d'impact sur la formule et engendrera beaucoup de clauses courtes. Cela dans le but de rencontrer rapidement des conflits et ainsi diminuer la profondeur de l'arbre.

5.2.2 Traitements locaux

Pendant le calcul de BSH, MTSS effectue certains traitements locaux tels que le picking, le pickdepth et le pickback (voir section 1.7.2 pour plus de détails).

5.2.3 Structures de représentation

Les structures utilisées par MTSS pour représenter la formule sont explicites. MTSS n'utilise pas de structures paresseuses car elles donnent de mauvaises performances avec l'heuristique choisie. De plus, les structures paresseuses sont utilisées dans les solveurs destinés à résoudre les formules industrielles, or l'implémentation actuelle de MTSS résout les formules aléatoires.

5.3 Gestion mémoire spécifique

Lors de l'optimisation mémoire d'une application séquentielle, les variables souvent accédées par l'algorithme sont regroupées en mémoire afin de minimiser le nombre de chargements depuis la mémoire centrale vers la mémoire cache du micro-processeur. Dans un contexte multi-cœurs, il en va tout autrement. En effet, soit d une donnée chargée dans la mémoire cache du cœur C_i. Si d est modifiée par le cœur de calcul C_j ($j \neq i$), alors la mémoire de C_i devra être mise à jour sans savoir si C_i accédera à nouveau à d. Ce phénomène a été mis en lumière dans [JK07], vous y trouverez plus de détails que ceux donnés ici.

5.3.1 Généralités

MTSS a donc une gestion mémoire qui n'est pas laissée au hasard. Au lieu de permettre au système de placer les variables en mémoire centrale, après avoir lu le fichier du problème à résoudre, MTSS détermine quelle quantité de mémoire sera nécessaire pour :

- Objets partagés en lecture seule
- Objets partagés en lecture et écriture
- Objets privés pour chaque processus

MTSS alloue des blocs mémoire assez grands pour contenir tous les objets d'un même ensemble et fait en sorte que ces ensembles ne se chevauchent pas sur la même page mémoire en allouant un peu d'espace supplémentaire et en alignant le début des variables sur une page mémoire. Il faut savoir que la mémoire des ordinateurs est paginée, c'est-à-dire que l'on n'y accède pas octet par octet mais page par page (le nombre d'octets enregistrés par page dépend du matériel utilisé). Donc, si un octet o est requis par un cœur de calcul C_i et que o n'est pas dans sa mémoire cache, alors

le système charge l'ensemble de la page contenant o dans la mémoire cache de C_i. Ce mécanisme est appelé un défaut de cache.

Pour chaque mémoire privée d'un processus, MTSS réitère le principe de cloisonnement pour éviter que les variables privées de C_i ne soient sur une page mémoire contenant des données privées de C_j.

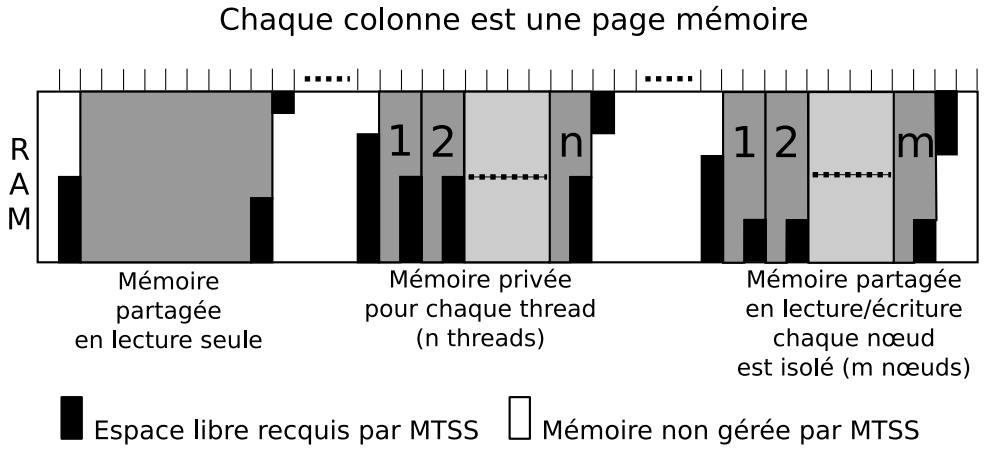


FIGURE 5.1 – Cloisonnement des types de mémoire

Cette gestion particulière est mise en place car nous souhaitons travailler sur des architectures de type CC-UMA voire CC-NUMA. Le fait que ces architectures soient *Cache Coherent* facilite la programmation mais provoque quelques soucis au niveau des défauts de cache, en particulier pour les problèmes combinatoires tels que SAT. D'un point de vue calcul pur, SAT n'est pas difficile à gérer pour les processeurs, la difficulté est le nombre gigantesque d'accès aléatoires en mémoire. Même si un défaut de cache ne représente finalement pas un long temps d'attente pour les processeurs, ce temps perdu multiplié par le très grand nombre d'accès en mémoire a de lourds impacts sur les performances globales. Un test a été réalisé (section 6.3) dans le but de vérifier cette affirmation.

5.3.2 L'arbre de guidage

L'arbre de guidage est parcouru par les pauvres pour détecter des nœuds pendants dans l'arbre et y calculer des tâches ou le nœud suivant. La partie à l'extrême gauche de cet arbre est développée par le riche. Tous développent simultanément le même arbre, mais chaque nœud est développé par un seul processus. Autrement dit, l'arbre dans sa globalité est un objet parallèle, mais chaque nœud durant son calcul doit être privé pour minimiser les défauts de cache engendrés par la cohérence mémoire. C'est pourquoi chaque nœud de l'arbre dans MTSS est isolé des autres. Aucune page mémoire ne contient d'information portant sur plusieurs nœuds. Les pauvres doivent pouvoir se positionner sur un nœud pour en calculer le suivant, mais si il fallait remettre à jour toutes les structures, ce serait trop coûteux en temps. Dans MTSS, chaque nœud

de l'arbre est capable de conserver une trace de l'instanciation courante pour que les pauvres ainsi que le riche puissent se mettre à jour en temps constant (contrairement au principe du chemin de guidage qui impose de simplifier la formule avec le chemin reçu). Pour le riche, cette mise à jour n'est faite que lorsqu'il effectue un retour arrière dans l'arbre après avoir trouvé une branche insatisfaisable et qu'un sous-arbre de guidage a été commencé au niveau du nœud i sur lequel il devait remonter. À cet instant, le riche effectue un parcours main gauche du sous-arbre A_i enraciné à la droite de i et copie dans son contexte privé les informations contenues au niveau du nœud j, ce dernier étant le nœud pendant le plus à gauche du sous-arbre A_i. Quant aux pauvres, deux cas s'offrent à eux. Soit ils commencent un sous-arbre de guidage, alors ils appliquent le principe du chemin de guidage sur le chemin du riche. Ils mettent à jour la formule de départ en fonction des choix opérés par le riche puisque le riche n'a pas pris le temps de déposer un contexte dans l'arbre (nous souhaitons que le riche soit le moins concerné possible par le surcoût engendré par le parallélisme). Soit ils travaillent dans un sous-arbre de guidage, et par conséquent, un pauvre a déjà laissé un contexte de calcul dans les nœuds. Dans ce cas, ils commencent toujours leur travail en copiant dans leur contexte privé celui contenu dans le nœud i précédent celui qu'ils doivent calculer. Une fois le travail terminé, ils déposent le nouveau contexte dans le nœud j ajouté à droite ou à gauche de i dans l'arbre de guidage.

L'arbre de guidage est donc une structure assez lourde en mémoire. Pour atténuer son importance, le nombre de nœuds développables simultanément par les pauvres est limité empiriquement à 80% du nombre de variables de la formule multiplié par le nombre de pauvres demandés. En réalité, en mémoire centrale, l'arbre de guidage n'est pas un simple tableau, mais une réserve de nœuds pointant vers son père et ses fils.

Il n'est pas rare de rencontrer un sous-arbre de guidage qui ne contienne pas de solution. Dans ce cas, toutes les feuilles ont conduit à un conflit. Le but des processus pauvres est de donner le maximum d'information au processus riche, il serait donc pénalisant que le processus riche ait à explorer entièrement un sous-arbre de guidage pour finalement découvrir que toutes les feuilles sont déjà calculées. Il est plus efficace de mettre l'information le plus tôt possible dans l'arbre. Dans le cas d'un sous-arbre sans solution, l'idéal est de faire remonter l'information jusqu'au nœud appartenant au chemin du riche. Il y a ici un double avantage : le processus riche n'explore pas un sous-arbre en vain et la consommation mémoire est réduite puisque les nœuds du sous-arbre en question seront libérés pour d'autres calculs. MTSS gère un *backtrack* pouvant être effectué par plusieurs processus à la fois grâce à la gestion d'un verrou spécifique à chaque nœud.

À l'image du chemin de guidage, l'arbre de guidage distribue le travail. Toutefois, l'équilibrage de charge est naturellement géré dans un arbre de guidage, il est inutile de recourir à une politique particulière. Le riche n'a pas besoin d'être équilibré puisque son comportement est celui d'un solveur séquentiel. Les processus pauvres, tant qu'ils sont utilisés pour calculer des tâches parallèles à grain fin sont naturellement équilibrés puisqu'ils travaillent sur des éléments très petits d'un arbre de recherche, il est inutile de devoir stopper certaines tâches pour les subdiviser.

5.4 Synchronisation

Pour que le programme MTSS soit performant sur une machine multi-cœurs, en sus de la gestion mémoire qui doit être finement réglée, un autre point sensible est la synchronisation entre processus. En effet, lorsque plusieurs unités de calcul accèdent à un même objet parallèle, il est nécessaire de mettre en place des verrous. Ces verrous sont utilisés lors des échanges par l'intermédiaire de l'arbre de guidage. Avant d'effectuer un calcul à un nœud, tout processus doit verrouiller le nœud le temps d'indiquer qu'il travaille sur celui-ci et peut ensuite le déverrouiller.

5.4.1 Échange d'information

Un des objectifs que nous nous sommes fixés pendant l'élaboration de MTSS était d'avoir un processus riche le plus proche possible d'un DLL classique, sans que le parallélisme vienne perturber son exécution. Pour cela, nous avons réduit l'échange d'information concernant le riche seulement lorsqu'il effectue un *backtrack* dans l'arbre. Pour les processus pauvres, la collaboration est beaucoup plus forte : ils s'échangent de l'information avant et après chaque tâche qu'ils exécutent. Cet échange d'information s'effectue par l'intermédiaire de l'arbre de guidage. L'arbre de guidage est capable de conserver un contexte de calcul entier par nœud : si un processus veut reprendre la suite du calcul à un nœud donné, il lui suffit de remplacer son contexte de calcul par celui enregistré dans l'arbre. Ainsi, le temps de mise à jour du contexte de calcul est constant pour une exécution donnée. Le changement de contexte n'est pas la seule information échangeable par l'arbre de guidage. Selon les aptitudes données aux pauvres, il peut y avoir divers types d'informations échangeables. Bien entendu, lorsqu'un sous-arbre est totalement exploré, la valeur de ce sous-arbre est connue au niveau du nœud. Le riche peut simplement lire ce résultat et continuer le *backtrack* à un niveau supérieur.

5.4.2 Principe de seuil

Cet échange d'information est constant, il est clairement plus coûteux en temps qu'une seule simplification dans la formule, mais cette copie mémoire est plus rapide qu'un nombre n de simplifications, n dépend du matériel utilisé et de la formule traitée. Ainsi, il n'est pas aisé de déterminer le nombre de simplifications pour lequel il est plus rentable de copier plutôt que de simplifier. Il est cependant évident que si le nombre de simplifications est très petit, il n'est pas intéressant de copier. C'est pourquoi un principe de seuil a été mis en place dans MTSS. Les pauvres ne dépassent pas une limite de profondeur dans le chemin du riche. Cette limite ne s'applique pas dans les sous-arbres de guidage. Le seuil a été placé empiriquement à la moitié de la hauteur maximum estimée de l'arbre de recherche. Cette hauteur dépend de l'heuristique de branchement utilisée. Pour BSH, la hauteur maximum estimée vaut un dixième du nombre total de variables, le seuil est donc fixé par défaut à 5% du nombre de variables dans la formule. Ce principe est symbolisé par la figure 5.2. Toutefois, ce seuil est modifiable par option.

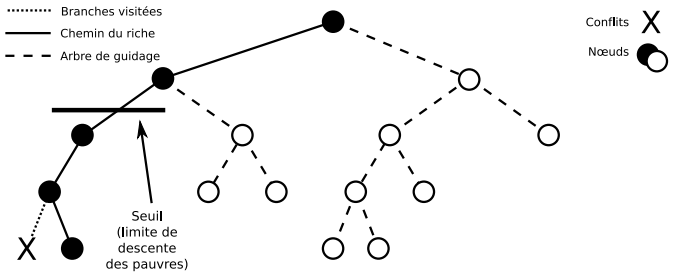


FIGURE 5.2 – Principe de seuil

5.4.3 Échange de rôle

Lorsque deux pauvres souhaitent travailler simultanément sur le même nœud, un seul sera en mesure de le faire. Le pauvre qui n'aura pas eu le droit de verrouiller le nœud pourra continuer sa recherche de travail et ne sera pas bloqué. Toutefois, cela est différent pour le processus riche qui doit conserver un parcours ordonné dans l'arbre de recherche afin d'assurer la complétude de l'algorithme. Dans le cas où il doit travailler sur un nœud en cours de calcul par un pauvre nommé P_i, le riche échange sa fonction avec le pauvre P_i. Le riche est donc absent du processus de résolution, mais lorsque P_i finit son calcul, il peut lire qu'il est le nouveau riche de MTSS. Le processus riche était donc virtuellement présent par le travail effectué par P_i. P_i adopte alors le comportement normal du riche, à savoir : explorer l'arbre de recherche en suivant un parcours main gauche, et développer sans en avoir conscience le chemin de guidage de MTSS qui permet la création de l'arbre de guidage par les pauvres. Techniquement, cela a mérité de mettre en place un gestionnaire des *threads* en amont des procédures pauvres et riche. L'identité du riche est tout simplement sauvegardée dans un entier. Une première version récursive a montré des performances en retrait. La récursivité alourdit la pile d'appels qui est un élément clé des *threads*. Il est important d'y prendre gard.

5.5 Les processus pauvres

La définition donnée en section 4.3.3 permet de nombreuses possibilités. Aujourd'hui MTSS se contente du minimum des tâches nécessaires pour la construction de l'arbre de guidage (sections 5.5.1 et 5.5.2) mais aussi de capacités destinées à rendre parallèle un programme séquentiel (section 5.5.4) en lui confiant le calcul d'un sous-arbre ou d'utiliser une heuristique de branchement externe à MTSS (section 5.5.5). Nous avons donc implémenté les tâches les plus extrêmes possibles. Il reste beaucoup de perspectives de développements à MTSS, notamment en lui ajoutant des traitements à exécuter à chaque nœud, sous la forme de fonctions incluses dans MTSS pour de meilleures performances ou sous la forme d'appels externes pour une meilleure réutilisation de l'existant.

5.5.1 Créer la racine d'un sous-arbre de guidage

Les pauvres parcourent le chemin du riche. Si la branche droite d'un nœud n n'est pas encore explorée, alors le pauvre reprend la formule de départ et la simplifie par les mêmes affectations que le riche avant lui (principe du chemin de guidage). Pour le nœud n, il simplifie la formule par l'affectation de la variable de n à la valeur opposée de celle choisie par le riche. Il en résulte une sous-formule qui est ensuite déposée dans l'arbre de guidage (schéma en figure 5.3). En sus de cette étape élémentaire, le pauvre calcule la valeur de la prochaine variable de branchement, qui est elle aussi déposée dans l'arbre de guidage.

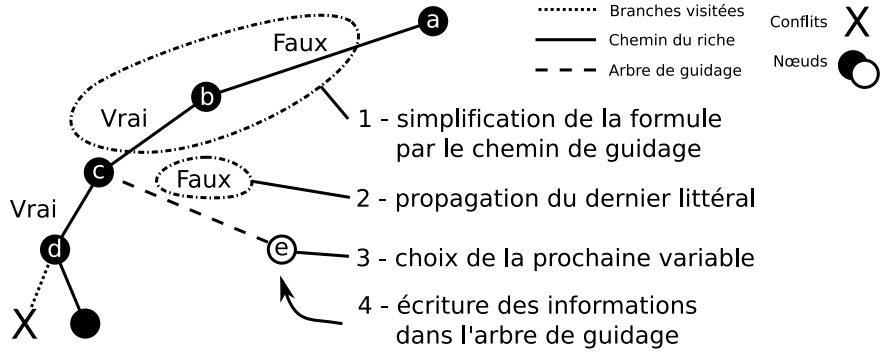


FIGURE 5.3 – Étapes de création d'une racine de sous-arbre de guidage

Soit i, le nœud sur lequel un pauvre souhaite ouvrir un sous-arbre de guidage à droite. Le temps de mise à jour de la formule par un pauvre (la lecture des littéraux du chemin du riche jusqu'au nœud i) pour qu'il puisse ouvrir un sous-arbre de guidage n'est pas considéré comme un travail de riche sur i. Cela signifie que si le riche doit *backtracker* au niveau du nœud i, alors il le fait sans attendre la fin de calcul du chemin de guidage du pauvre et sans lui donner la main en tant que riche dans l'arbre. Le travail effectué par le pauvre a été fait en vain. Toutefois, les pauvres sont capables de détecter qu'une telle action est en cours de calcul afin de ne pas la faire à plusieurs en simultané puisqu'un seul pourra ouvrir le sous-arbre de guidage.

5.5.2 Développer un sous-arbre de guidage

Les pauvres parcourent le chemin du riche de manière cyclique. À chaque nœud dont le sous-arbre droit n'a pas encore été visité par le riche, les pauvres explorent le sous-arbre de guidage en suivant un parcours main gauche. Ils développent le premier nœud disponible, en cas de plusieurs pauvres voulant travailler sur le même nœud, seul le premier à demander le verrou sur ce nœud pourra travailler, les autres verront ce nœud en cours de calcul et continueront leur parcours main gauche. Ce travail s'effectue en trois étapes (voir schéma en figure 5.4) :

– Préparation : copie du contexte de calcul depuis l'arbre de guidage vers le contexte local.

– Calcul : simplification de la sous-formule locale par le littéral du nœud précédent dans le cas d'une branche gauche ou par la valeur opposée dans le cas d'une branche droite. Puis choix heuristique du prochain littéral à propager. Ajout du nœud contenant ce littéral dans l'arbre de guidage.
– Passage d'information : copie du contexte de calcul depuis la mémoire privée vers celui de l'arbre de guidage.

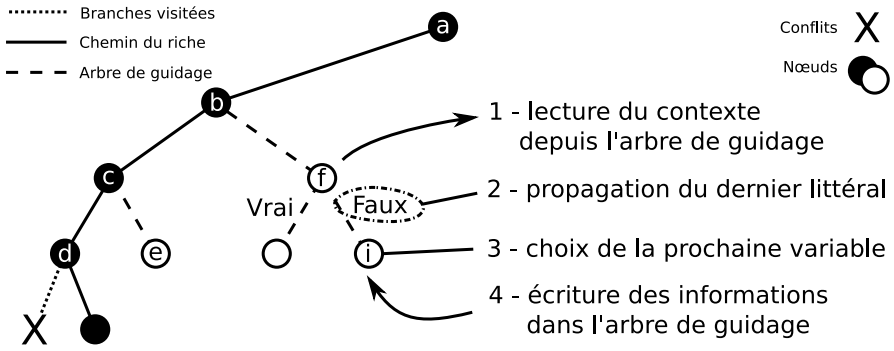


FIGURE 5.4 – Étapes du prolongement d'un sous-arbre de guidage

5.5.3 Exemple d'exécution de MTSS

Voici maintenant un exemple d'exécution possible sur une formule insatisfaisable en utilisant 3 *threads*, donc 1 riche et 2 pauvres. Cet exemple montre le déroulement de l'algorithme avec les seules tâches pauvres détaillées jusqu'ici, à savoir la création d'une racine de sous-arbre de guidage et le développement de ces sous-arbres. Pour simplifier l'explication, les processus changent de nœud presque à chaque étape, mais il faut bien garder à l'esprit qu'en réalité, les processus ne sont pas synchrones. Le processus riche est nommé *Rich Thread* et est reconnaissable par la couleur verte. Les processus pauvres sont le bleu et le rouge, ils sont nommés *Poor Threads*. Les carrés entourant les nœuds de l'arbre de recherche représentent la présence d'un processus en cours de calcul. Les numéros des nœuds ne représentent pas les variables choisies mais servent simplement à les identifier. Les croix représentent un conflit rencontré pendant le processus de recherche.

Première étape (figure 5.5), le processus riche travaille sur le premier nœud de l'arbre, les pauvres ne peuvent trouver de tâche à remplir, ils attendent.

FIGURE 5.5 – Exemple MTSS, étape 1

À la seconde étape (figure 5.6), le riche traite un deuxième nœud, son parcours est suivi par une flèche verte et son chemin est identifiable par une plus grande épaisseur.

Puisque la racine a été calculée par le processus riche, le sous-arbre droit est disponible pour que les processus pauvres y travaillent. La racine de ce sous-arbre droit est prise en charge par le processus pauvre numéro 2.

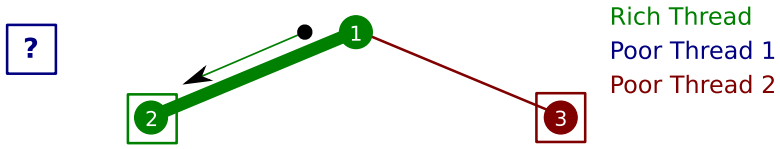


FIGURE 5.6 – Exemple MTSS, étape 2

Le riche poursuit sa recherche de solution (figure 5.7). Les pauvres recherchent des tâches à exécuter dans l'arbre de guidage. Le processus pauvre 1 a trouvé une tâche à réaliser sur le chemin de guidage du riche. Le processus pauvre 2 développe un nœud à la suite de sa précédente tâche.

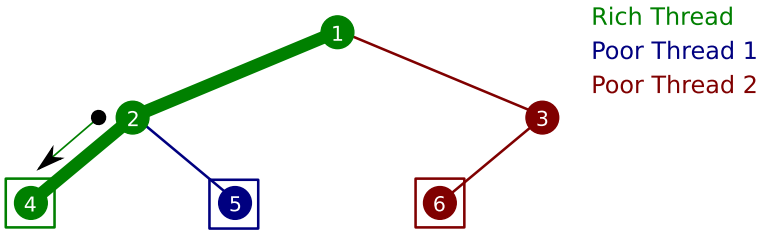


FIGURE 5.7 – Exemple MTSS, étape 3

Encore une fois (figure 5.8), le riche continue de développer son arbre tandis que les pauvres sautent littéralement de sous-arbre de guidage en sous-arbre de guidage à la recherche d'une tâche à effectuer. À cette étape, tous les processus ont rencontré un conflit. Les processus pauvres remontent l'information d'un sous-arbre sans solution jusqu'à sa racine afin d'accélérer la lecture du processus riche. Sur cette étape, on peut voir le principe de seuil (trait noir discontinu) sur le chemin du riche. Le seuil empêche les pauvres de descendre plus bas dans l'arbre. Les nœuds sous cette limite ne seront développés que par le processus riche.

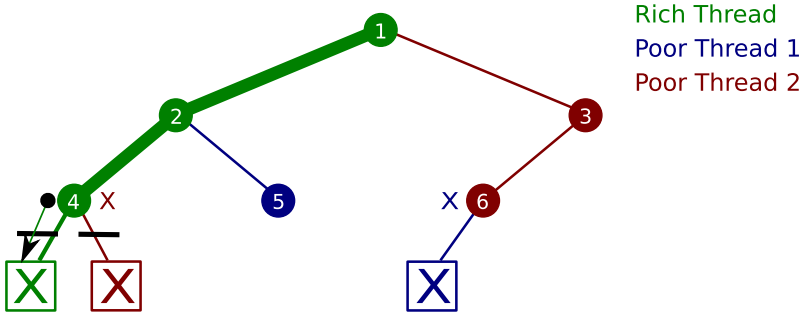


FIGURE 5.8 – Exemple MTSS, étape 4

Les pauvres continuent de développer ensemble l'arbre de guidage (figure 5.9). Le riche *backtrack* sur le nœud numéro 4 et lit sans plonger dans le sous-arbre droit que celui-ci ne contient aucune solution. Le *backtrack* du riche continue immédiatement jusqu'au nœud numéro 2, puis le riche prend en charge la continuation du nœud le plus à gauche du sous-arbre droit du nœud numéro 2 (le nœud numéro 5). Ici le riche n'a pas eu besoin de recalculer ses structures, mais a repris celui de l'arbre de guidage préalablement déposé au nœud 5 par le processus pauvre numéro 1.

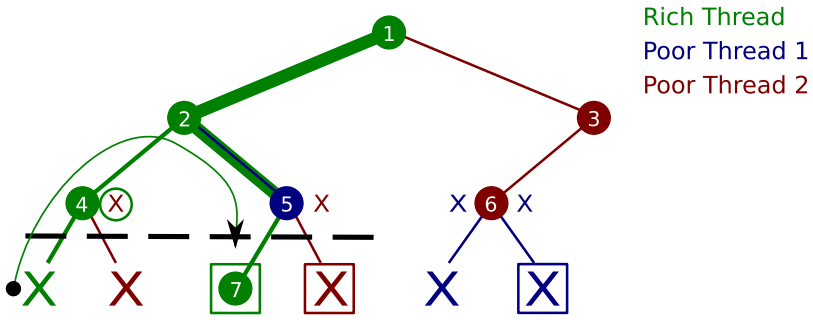


FIGURE 5.9 – Exemple MTSS, étape 5

Sur la figure 5.10, le pauvre numéro 1 a remonté l'information de conflit au nœud 6 puis jusqu'à la gauche du nœud 3. Dans le même temps, l'autre pauvre continue de développer le sous-arbre de guidage à la droite de la racine. En effet, à cause du seuil sur la branche riche, le sous-arbre gauche de la racine ne contient plus de nœuds disponibles pour les pauvres. Le nœud 7 devra être totalement développé par le processus riche.

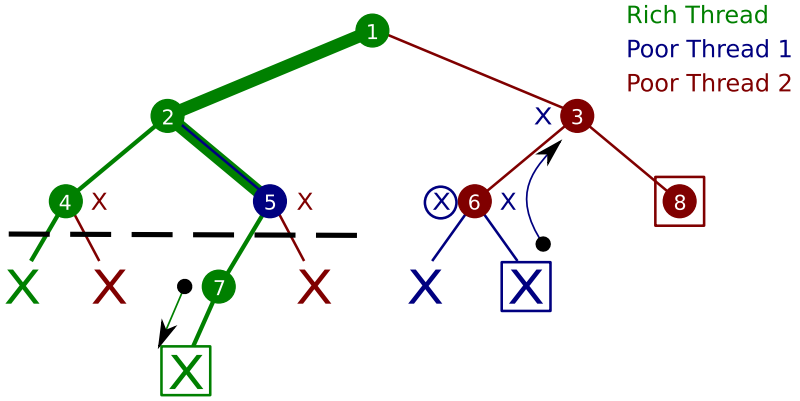


FIGURE 5.10 – Exemple MTSS, étape 6

Figure 5.11, tandis que le processus riche s'occupe de l'autre branche du nœud 7, les pauvres continuent de développer le sous-arbre de guidage de droite (fils du nœud 8).

À l'étape 8 (figure 5.12), le riche retourne jusqu'à la racine car il n'a trouvé aucune solution dans le sous-arbre gauche du nœud 1. le *backtrack* est direct grâce aux informations laissées par les pauvres au niveau du nœud 5. De leur côté, les pauvres continuent

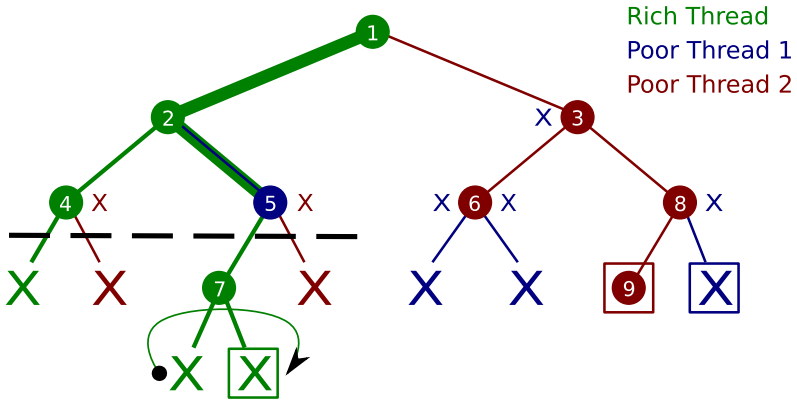


FIGURE 5.11 – Exemple MTSS, étape 7

de travailler. Ils peuvent descendre plus profondément dans l'arbre de recherche que le niveau du seuil, car le seuil n'est défini que pour le chemin du riche.

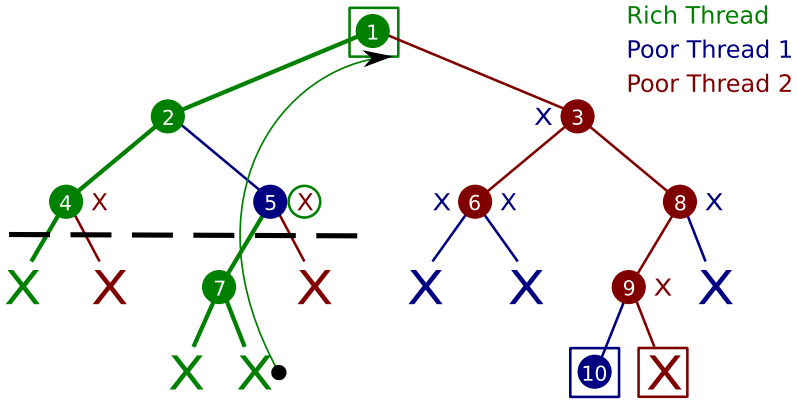


FIGURE 5.12 – Exemple MTSS, étape 8

Ensuite (figure 5.13), le riche part directement travailler de la droite du nœud 7 jusqu'à la gauche du nœud 10 grâce à toutes les tâches effectuées par les processus pauvres. Le nœud 10 est plus profond que le seuil et toutes les autres tâches sont déjà calculées. Les 2 processus pauvres ne sont donc plus requis.

Le riche calcule la droite du nœud 10 (figure 5.14).

Pour terminer cet exemple, le riche peut accéder directement à la racine depuis le nœud 10 car les sous-arbres droits des nœuds 8 et 9 sont déjà déclarés insatisfaisables par les processus pauvres. Avec l'aide de ces derniers, le processus riche peut conclure que la formule n'a pas de solution.

Comme dit plus haut, les processus pauvres équilibrent les tâches qui leur sont destinées entre eux naturellement puisqu'ils parcourent tous l'arbre de guidage dans sa globalité et y appliquent un traitement très petit en temps de calcul.

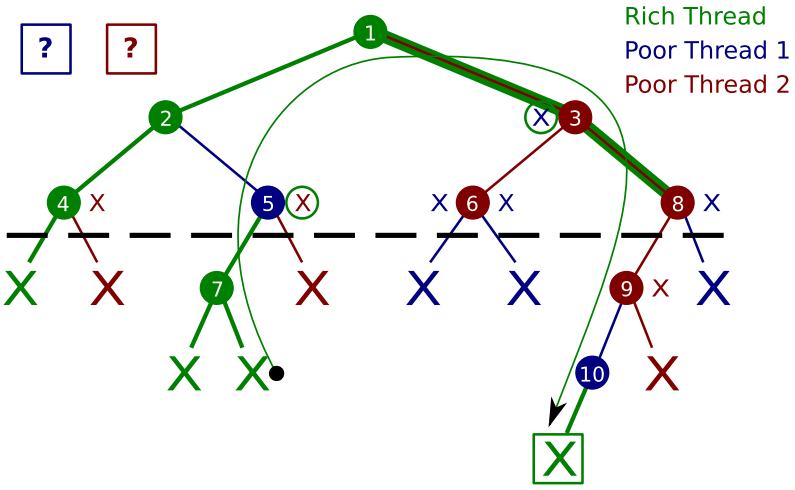


FIGURE 5.13 – Exemple MTSS, étape 9

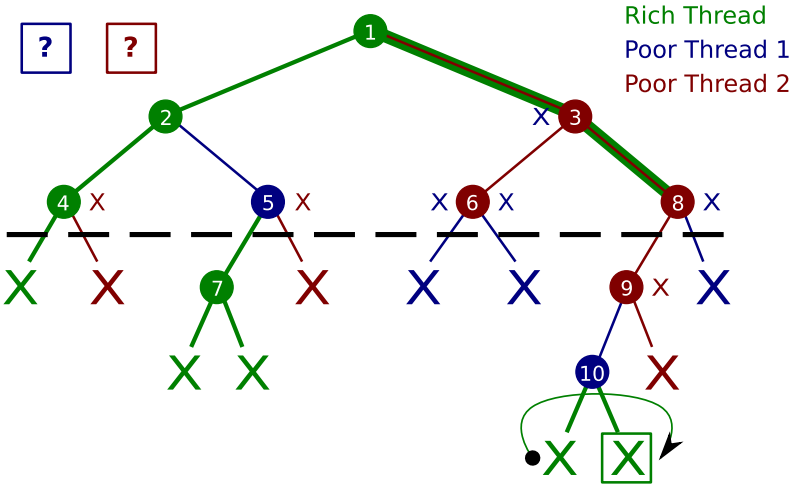


FIGURE 5.14 – Exemple MTSS, étape 10

5.5.4 Lancer un solveur externe à MTSS

5.5.4.1 Motivations

Bien évidemment, le plus efficace en termes de performances pour étendre les capacités de MTSS, par exemple pour les processus pauvres, est de développer une fonction optimisée pour ses structures. Toutefois, nous faisons le constat qu'il existe aujourd'hui énormément de solveurs SAT séquentiels très performants (voir chapitre 1) que nous avons pensé utile et intéressant pour la communauté de chercheurs que MTSS soit capable de paralléliser leurs approches. Beaucoup de solveurs séquentiels offrent d'excellentes performances grâce à des innovations qui leur sont propres et sont encore maintenus aujourd'hui, contrairement à la majorité des solveurs parallèles. Pourquoi devoir

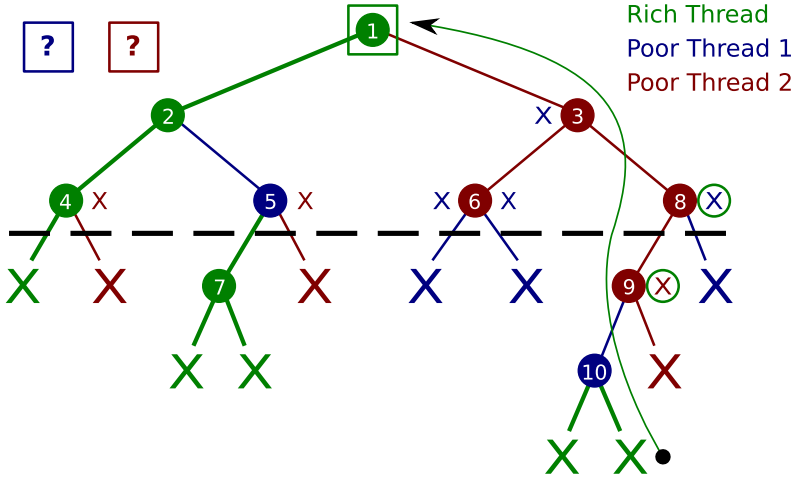


FIGURE 5.15 – Exemple MTSS, étape 11

développer une version parallèle de toutes les versions des solveurs séquentiels ? Cela ne ferait qu'alourdir la tâche des chercheurs : après une implémentation réussie d'un nouveau traitement dans un solveur séquentiel, il faudrait l'implémenter dans un solveur parallèle. Bien sûr, ce travail doublé est la conséquence d'un travail séquentiel que l'on souhaiterait paralléliser, alors qu'une version parallèle unique serait suffisante. Mais la réalité actuelle de l'état de l'art SAT montre que le nombre de solveurs séquentiels est beaucoup plus grand que le nombre de solveurs parallèles. Aujourd'hui la majorité des chercheurs SAT ne programment qu'en séquentiel. De plus, la programmation parallèle demande quelques notions que tout le monde n'a pas ou ne souhaite pas acquérir. Pour ces raisons, et parce que MTSS a la particularité d'offrir un parallélisme à plusieurs niveaux de granularité, nous avons ajouté une tâche à l'éventail des tâches disponibles aux processus de MTSS. MTSS donne la possibilité de paralléliser facilement, et parfois sans modification, un solveur SAT séquentiel. Il est aussi possible de paralléliser un solveur déjà parallèle mais qui serait limité en nombre de cœurs utilisés.

5.5.4.2 Principe

Nous appelons mutation le moment où le solveur externe se substitue à MTSS. Cette mutation a lieu lorsque les processus de MTSS atteignent la profondeur définie par l'utilisateur. Avant cette limite, MTSS conserve un comportement normal de développement de l'arbre de guidage. Cette tâche étant destinée à paralléliser un solveur externe, MTSS doit minimiser son impact sur la recherche de solution. MTSS sert simplement à générer des sous-formules puis à les faire résoudre par le solveur parallélisé. C'est pourquoi cette tâche a la particularité d'être également exécutée par le riche. La figure 5.16 montre ces éléments.

N'ayant pas la capacité d'adresser le même élément en mémoire centrale puisque MTSS lance un programme externe, l'arbre de guidage n'a plus de sens pour les solveurs externes. Le principe de distribution du travail est celui du chemin de guidage, à savoir :

MTSS fournit un chemin au solveur externe afin qu'il puisse travailler sur une sous-formule associée au sous-arbre qui lui a été assigné. Ainsi, contrairement aux tâches pauvres qui développent l'arbre de guidage, qui sont des tâches de grain fin, le grain mis en place pour cet outil de parallélisation est gros car il correspond au calcul d'un sous-arbre. Dans ce cas, il peut être utile d'implémenter une politique d'équilibrage de charge comme il est nécessaire de la faire lorsque l'on utilise les chemins de guidage. En effet, il est probable que la difficulté de résolution d'une formule soit concentrée dans un seul sous-arbre, et dans ce cas, ce sous-arbre serait plus long à parcourir que les autres. Le processeur auquel un tel sous-arbre serait affecté devrait continuer à calculer alors que les autres processeurs resteraient oisifs. Malheureusement, dans MTSS, nous n'avons pour le moment implémenté aucune politique d'équilibrage de charge explicite pour les tâches à gros grain, mais il n'est pas impossible de le faire. Dans ce cas, il serait nécessaire de récupérer le chemin courant sur lequel s'exécute un solveur externe pour pouvoir le redécouper tel un chemin de guidage et redistribuer la charge parmi les processeurs disponibles. Un tel mécanisme alourdit la procédure de parallélisation de solveurs externes alors que nous souhaitons mettre en place un principe simple et efficace. Toutefois, nous avons séparé deux types de solveurs que nous proposons de paralléliser avec plus ou moins de modifications selon les performances que l'on souhaite obtenir.

Étant donné que les formules générées aléatoirement et celles tirées du monde réel sont résolues efficacement de manières différentes, nous devons adapter la recherche de solution au type d'instance en cours de résolution. Dans tous les cas, pour que le passage d'information puisse avoir lieu, il est nécessaire que le solveur externe parallélisé puisse intégrer une formule SAT au format DIMACS. De même, sa sortie doit être normalisée, nous avons utilisé la sortie standardisée par le format DIMACS. C'est-à-dire que toute ligne doit commencer par le caractère 'c', sauf la ligne de résultat qui commence par 's' suivi de « Satisfiable » ou « Unsatisfiable » et la ou les lignes de valeurs de variables doivent commencer par 'v' suivi des littéraux propagés dans la formule.

5.5.4.3 Solveurs pour formules générées aléatoirement

Si le solveur parallélisé est conforme aux standards d'entrée et sortie donnés ci-dessus, la parallélisation est ici immédiate. Il suffit de lancer MTSS avec les options adéquates et le solveur sera lancé sur chaque sous-arbre à partir de la profondeur désirée. Il n'est pas nécessaire de gérer l'intéraction avec MTSS pour recevoir le chemin de guidage car MTSS aura écrit une nouvelle formule ayant subi les simplifications correspondant au chemin de guidage désignant le sous-arbre sur lequel le solveur externe doit travailler. Vous trouverez un ensemble de tests dans le chapitre 6.

5.5.4.4 Solveurs industriels

Les particularités des formules dites industrielles sont leur taille ainsi que la logique contenue dans celles-ci. L'heuristique intégrée à MTSS est trop coûteuse en temps lorsque le nombre de variables et de clauses sont grands. Pour paralléliser de tels solveurs,

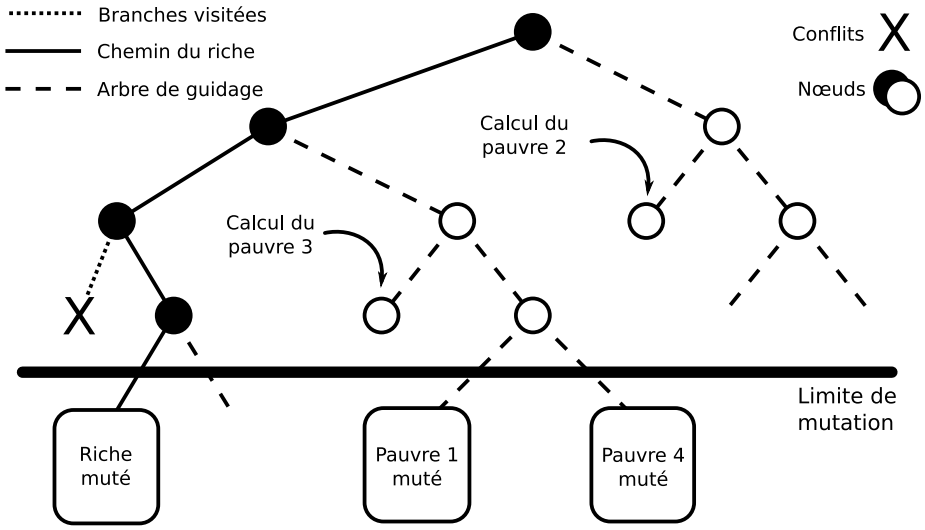


FIGURE 5.16 – Parallélisation de solveurs externes

nous avons intégré l'heuristique naïve MOMS afin de faire des choix rapides en haut de l'arbre. Pour assurer de bonnes performances à MTSS, il fallait le doter d'un mécanisme d'échange de clauses. MTSS n'apprend pas de clauses à partir des conflits rencontrés mais implémente un système d'échange de clauses entre chacun de ses processus et les exécutions du solveur externe. Une fois que les clauses sont ajoutées à une base connue par tous les processus, ils sont capables de les distribuer parmi les exécutions du solveur externe. La base commune de clause est gérée de façon assez simple pour l'instant, par exemple le compactage de la base se fait de manière arbitraire : 1 clause sur 2 est effacée. Malgré une approche simplifiée, sans avoir implémenté de mécanisme complexe d'échange de clauses pour un solveur séquentiel, il est dorénavant capable de le faire en parallèle grâce à l'utilisation de MTSS.

L'échange de clauses est soumis à certaines règles imposées par MTSS. La longueur maximale des clauses partagées est de 8 littéraux (meilleure taille pour `Minisat 2`, empiriquement déterminée, lorsque cette limite est fixe [HJS09b]). Le partage de clauses requiert quelques modifications pour que MTSS et le solveur externe puissent coopérer. Nous essayons d'apporter une solution simple avec un ensemble d'outils rassemblés dans une bibliothèque nommée MTSS.h. Les modifications à apporter au solveur externe peuvent se résumer ainsi :

- Initialisation : après la lecture de la formule, le solveur doit récupérer le chemin qui lui permettra de simplifier la formule pour travailler sur le sous-arbre qui lui a été affecté. Ce principe est le même que celui du chemin de guidage dans un programme parallèle classique puisqu'il est impossible d'échanger l'arbre de guidage avec des programmes externes à MTSS.
- Partage : à chaque redémarrage du solveur externe (les solveurs modernes ont tous une politique de redémarrage), les nouvelles clauses apprises sont envoyées à MTSS puis celles apprises par ses exécutions concurrentes sont intégrées.

Les modifications à apporter dans ce type de solveur restent mineures compte tenu des avantages obtenus, à savoir : un solveur séquentiel qui est devenu parallèle.

5.5.4.5 Exemple de parallélisation de solveur avec MTSS

La manière de présenter cet exemple est la même que pour l'exemple précédent (voir section 5.5.3), mais ici, le trait noir discontinu n'est plus le seuil de profondeur maximum des pauvres mais la limite de mutation, cette limite est donc valable sur l'ensemble de l'arbre de recherche. Les gros carrés sous cette limite représentent le calcul externalisé par MTSS dans le solveur choisi par l'utilisateur.

À l'étape 1 (figure 5.17), on constate que l'algorithme a déjà commencé à ouvrir l'arbre de guidage puisque le comportement normal de MTSS est conservé en haut de l'arbre, avant la limite de mutation. Trois exécutions simultanées du solveur externe sont en cours car 3 processus de MTSS ont atteint cette limite. Après la propagation de la variable de choix, la sous-formule résultante est résolue par le solveur externe.

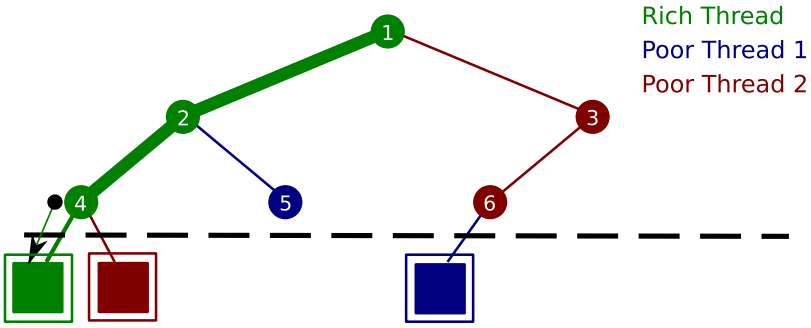


FIGURE 5.17 – Exemple de parallélisation, étape 1

À l'étape suivante (figure 5.18), le riche a terminé le sous-arbre gauche du nœud 4. Dans une exécution standard de MTSS, il chercherait à travailler sur le nœud le plus à gauche du sous-arbre de droite du nœud 4. Or, ce sous-arbre est en cours de calcul par le processus pauvre 2. L'information d'un pauvre est en cours de calcul et par conséquent pas encore disponible. Afin de ne pas perdre un temps précieux, il informe le processus pauvre 2, en laissant une information au niveau du nœud 4, que celui-ci deviendra le prochain processus riche.

Ensuite (figure 5.19), le processus riche de l'étape précédente (le vert), devient un processus pauvre et part explorer l'arbre de guidage à la recherche d'une tâche à effectuer. Ici, il a choisi de calculer le sous-arbre droit du nœud 6. Comme on le voit sur la figure, le processus le plus à gauche maintenant est le rouge. Il est donc bien le nouveau processus riche, même s'il ne le saura qu'après avoir fini son travail actuel. Ce principe d'échange de rôle n'est pas lié à l'utilisation d'un solveur externe mais existe aussi dans le cas d'une exécution classique de MTSS.

Encore une fois, un processus (le rouge) atteint la limite de mutation (figure 5.20). Ce processus était le riche, il n'avait donc pas le choix, il devait calculer le nœud le plus à gauche de l'arbre. Cette fois encore, ce nœud (gauche du nœud 5) n'était pas disponible

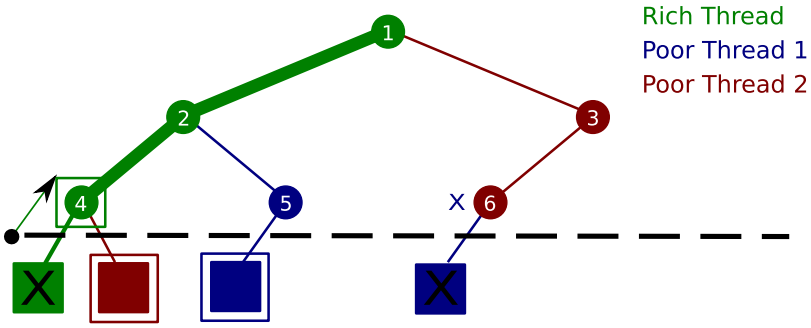


FIGURE 5.18 – Exemple de parallélisation, étape 2

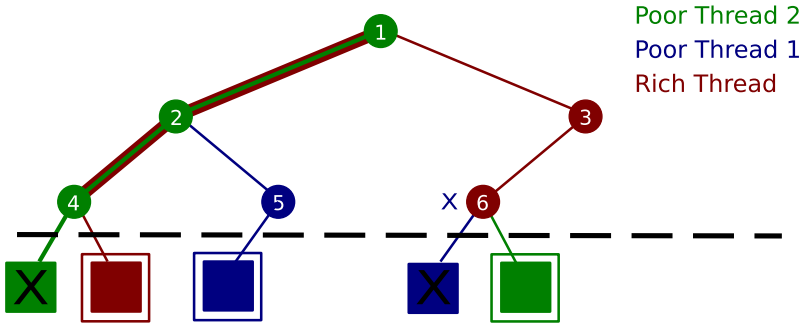


FIGURE 5.19 – Exemple de parallélisation, étape 3

car en cours de calcul par un processus pauvre. Le processus riche (le rouge), devenu processus pauvre, cherche donc une nouvelle tâche à effectuer. Le nouveau processus riche, le bleu, devra à son tour retrouver le nœud le plus à gauche pour garantir la complétude de l'algorithme.

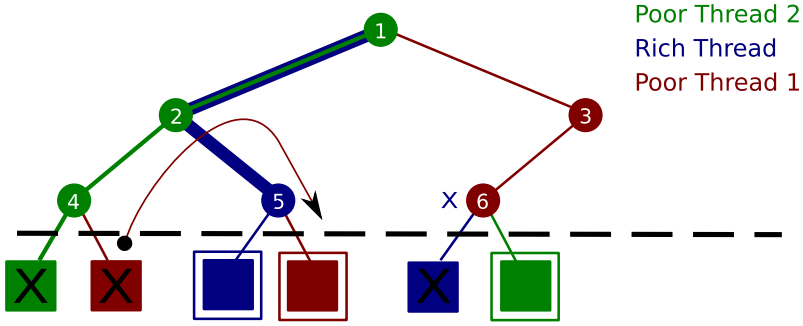


FIGURE 5.20 – Exemple de parallélisation, étape 4

5.5.5 Utiliser une heuristique de branchement externe à MTSS

MTSS permet également l'utilisation d'une heuristique de branchement externe. Le principe est le même que pour le solveur externe, mais au lieu de déclencher un solveur externe à partir d'une profondeur choisie, MTSS lance un solveur externe, ou un programme spécifique à chaque fois qu'un choix heuristique est à réaliser dans MTSS. Pour que cette aptitude fonctionne, il suffit que le programme externe puisse lire un fichier DIMACS et qu'il affiche ses informations au format DIMACS, sauf pour le choix qu'il aura fait. Il devra afficher par exemple "r -439" si il a décidé de choisir la variable 439 affectée négativement en tant que racine du sous-arbre qui lui aura été assigné. Étant donné que le nombre de nœuds est sujet a explosé de manière exponentielle, il y a énormément d'appels systèmes pour gérer un grand nombre de choix heuristiques externes. Dans de telles conditions, les performances de MTSS reculent de façon significative.

5.6 Conclusion

Comme nous le permettait la définition de MTSS, nous avons d'un côté implémenté un grain très fin afin de développer l'arbre de guidage, et de l'autre nous avons intégré à MTSS une tâche à gros grain par l'outil de parallélisation. L'arbre de guidage et sa construction fortement collaborative ont amené des difficultés techniques que nous avons résolues par une gestion mémoire bien maîtrisée et une synchronisation bien orchestrée. La gestion mémoire a pour but de minimiser les défauts de mémoire cache alors que la synchronisation millimétrée et l'échange de rôle réduisent les temps d'attente qui sont le goulot d'entranglement des performances pour les systèmes à mémoire partagée, au même titre que les messages le sont pour les systèmes à mémoire distribuée. En outre, MTSS apporte une solution de calcul externe, ce qui lui confère un réel intérêt pour la communauté qui pourra dorénavant facilement tirer partie des machines multi-cœurs à leur disposition. Pour des performances certainement moindres que de programmer de A à Z une version parallèle d'un solveur existant, mais pour un coût de parallélisation relativement nul, il est possible de rendre parallèle un solveur capable de n'exécuter qu'un seul processus. MTSS ouvre une voie vers un nouveau parallélisme dans la communauté SAT, d'autant plus que la majeure partie des solveurs parallèles de la littérature n'ont jamais réellement été maintenus. Les chercheurs préfèrent les versions séquentielles où la programmation d'une nouvelle heuristique ou d'un nouveau calcul est plus aisée. Nous espérons voir notre solveur utilisé par la communauté.

Chapitre 6

Résultats expérimentaux

6.1 Introduction

Ce chapitre fait une synthèse des résultats que nous avons obtenus : de la gestion mémoire spécifique (section 6.3) à l'utilisation de solveurs externes (section 6.6) en passant par l'utilisation normale de MTSS (section 6.5) afin de vérifier son efficacité dans un cadre multi-cœurs. Des explications sur les bonnes ou parfois moins bonnes performances sont données. Pour information, dans l'ensemble des tests présentés ici, tous les *threads* logiques que nous exécutons sont gérés par un cœur de calcul, il y a toujours assez de cœurs de calcul sur la machine qui exécute MTSS pour gérer l'ensemble de ses *threads*.

6.2 Protocole

Le super calculateur utilisé pour l'ensemble des *benchmarks* présentés ici est ROMEO II de l'Université de Reims Champagne-Ardenne. Je vous renvois au tableau 2.1 pour plus de détails sur cette machine parallèle. Le compilateur utilisé est icc dans sa version 10. Pour l'ensemble de ces tests, MTSS utilise des *threads* OpenMP.

6.3 Gestion mémoire

La figure 6.1 montre l'efficacité de MTSS en fonction de l'allocation mémoire utilisée. À l'époque de ce test, les deux versions du solveur étaient identiques, seule la gestion mémoire spécifique avait été remplacée par des allocations à l'aide de « *mallocs* » sans influence de notre part dans les adresses mémoire. Toutefois, ce test est ancien et les performances séquentielles de MTSS n'étaient pas celles d'aujourd'hui. Des différences d'efficacité sont donc à noter entre ce graphique et ceux plus récents que vous trouverez en section 6.5. L'information notable de cette figure est donc l'écart, et non l'efficacité. On constate que la version optimisée pour une exécution sur processeur multi-cœurs est entre 10 % et 15 % plus efficace que la version à base de *mallocs*. Il est donc évident que le cloisonnement des mémoires selon l'usage qui en est fait impacte les

performances de façon significative. Sur ce graphique, l'axe des ordonnées représente le pourcentage d'efficacité et l'axe des abscisses donne le nombre de *threads* lancés. Les formules résolues lors de ces tests étaient des 3-SAT aléatoires mais les deux versions du solveur ont résolu les mêmes formules.

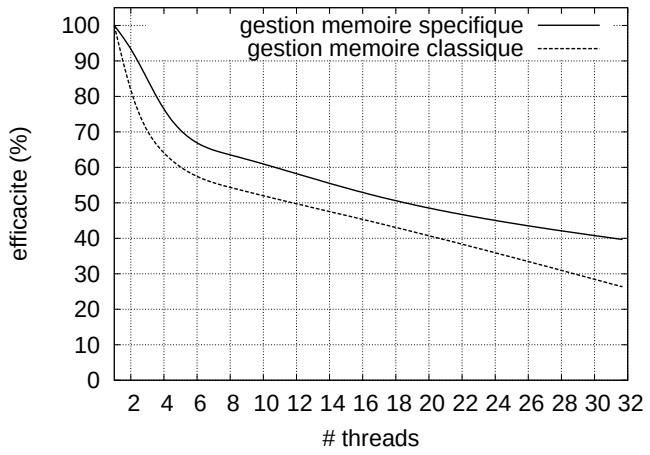
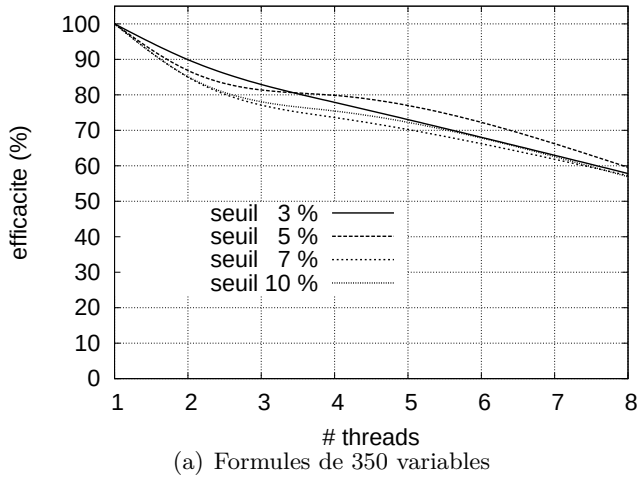


FIGURE 6.1 – Gains obtenus par la gestion mémoire spécifique

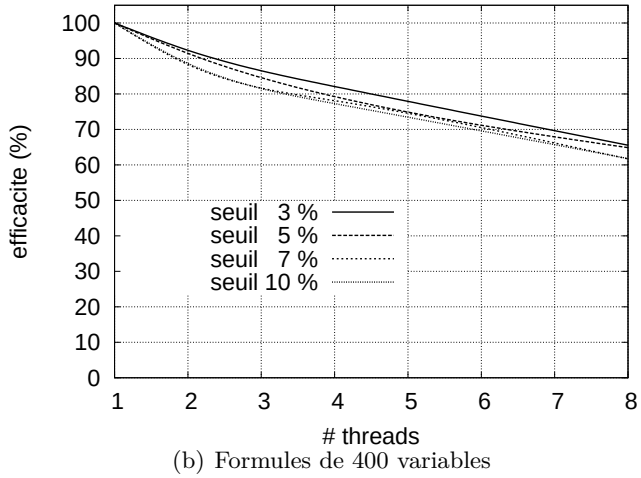
6.4 Seuil limite des pauvres

Afin de limiter des échanges trop nombreux près des feuilles portant sur peu d'information. Nous avons mis en place un seuil sur le chemin développé par le riche au-delà duquel les processus pauvres ne peuvent pas développer de sous-arbre de guidage. Le seuil s'exprime en nombre de décision (profondeur de l'arbre). Le niveau de seuil est empiriquement fixé puisqu'il dépend de l'heuristique de branchement choisie. Pour l'heuristique BSH, nous avons fixé ce seuil à 5% du nombre de variables dans la formule à résoudre. Les sous-arbres au-delà de cette limite sont très rapides à résoudre puisqu'ils sont composés de peu de nœuds. Cela permet aux processus pauvres de concentrer leur travail sur les sous-arbres plus importants et permet au riche de récolter beaucoup d'information lors d'un changement de contexte. Un *benchmark* a été réalisé avec une précédente version de MTSS qui ne pratiquait pas encore l'échange de rôle riche / pauvre. Ce test est montré sur les figures 6.2. Les formules utilisées pour ce *benchmark* sont toutes 3-SAT aléatoires au pic de difficulté et insatisfaisables. La valeur de seuil à 10% représente en réalité une absence de seuil car il est rare avec BSH de construire des chemins composés d'un nombre de décisions supérieur à 10% du nombre de variables dans la formule.

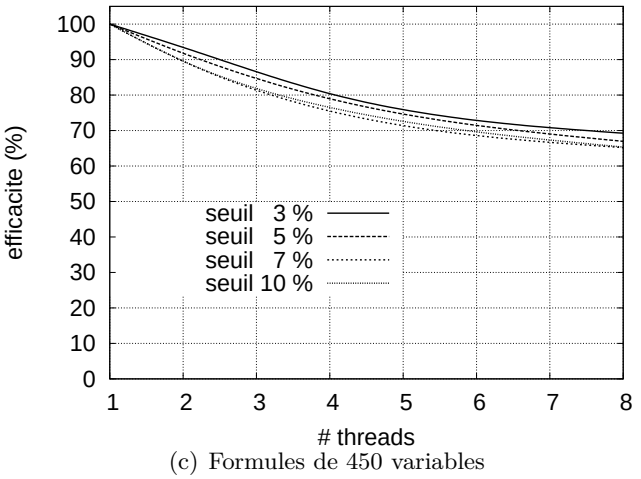
Il faut dorénavant relativiser ces résultats car depuis la capacité à s'échanger les rôles, les différences sont encore moins significatives, d'ailleurs, il est préférable de ne pas mettre le seuil trop haut dans l'arbre afin que les pauvres aient assez de sous-arbres de guidage à déployer. En d'autres termes, si le seuil est trop bas (en pourcentage du



(a) Formules de 350 variables



(b) Formules de 400 variables



(c) Formules de 450 variables

FIGURE 6.2 – Influence du seuil choisi sur différentes difficultés de formules

nombre de variables), les performances sont réduites alors que le seuil au delà de 10% est moins pénalisant que dans la version précédente de MTSS. Cela implique que les mauvaises performances vers 10% était certainement dûes en partie à l'attente que subissait le processus riche pour attendre l'information calculée par les processus pauvres.

6.5 Efficacité

L'efficacité relative de MTSS est montrée dans la figure 6.3. L'axe des ordonnées représente l'efficacité en pourcentage et celui des abscisses donne le nombre de *threads* exécutés. Les formules résolues pour ce test sont des formules 3-SAT aléatoires de 500 variables toutes insatisfaisables afin d'éviter des gains d'efficacité dûes à la découverte rapide d'une solution (accélération parfois super linéaire). Ainsi, l'efficacité donnée ici est la pire que MTSS puisse fournir. Le temps séquentiel moyen requis par MTSS pour résoudre une formule de ce type est de 5000 secondes. On constate que MTSS a une excellente efficacité en utilisant un seul *thread* pauvre (2 *threads* au total) : 93 %, et conserve une très bonne efficacité ensuite : 86 % sur 4 *threads* et 82 % sur 8. Sur 8 cœurs de calcul, MTSS résout chacune de ces formules en moyenne en 750 secondes (performances en figure 6.4). De plus, l'écart de temps sur la même formule n'excède pas quelques secondes, quelque soit le nombre de *threads* lancés. MTSS est donc véloce mais aussi robuste pour résoudre les formules aléatoires.

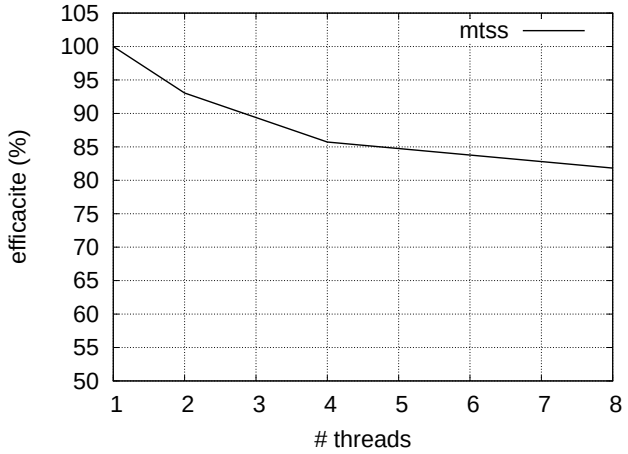


FIGURE 6.3 – Efficacité relative de MTSS jusqu'à 8 cœurs

6.6 Utilisation de solveurs externes

6.6.1 Solveurs pour formules générées aléatoirement

Les figures 6.5 et 6.6 donnent les résultats obtenus sur vingt formules 3-SAT générées aléatoirement au seuil de difficulté (section 1.8.2 pour plus de détails). Elles contiennent

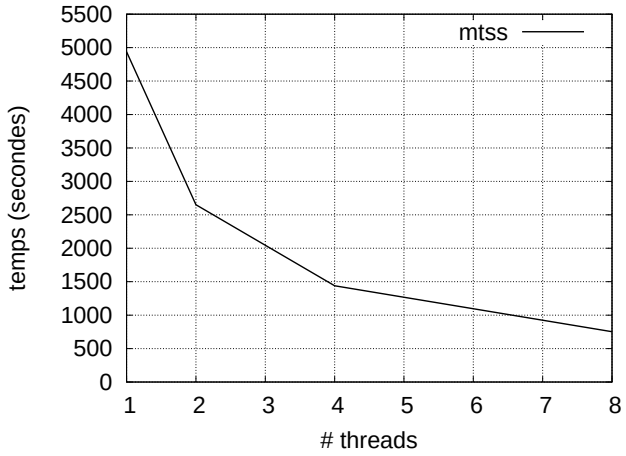


FIGURE 6.4 – Performances de MTSS jusqu'à 8 cœurs

toutes 500 variables et sont insatisfaisables pour éviter des gains super linéaires qui pourraient brouiller les résultats. Chaque courbe est la moyenne de plusieurs exécutions, et ce, pour différents nombre de *threads* (1, 2, 4 et 8). Les solveurs externes utilisés sont `kcnfs` et `march` [HvM06]. MTSS procédait à la mutation en solveur SAT externe à la profondeur 5 dans l'arbre de recherche.

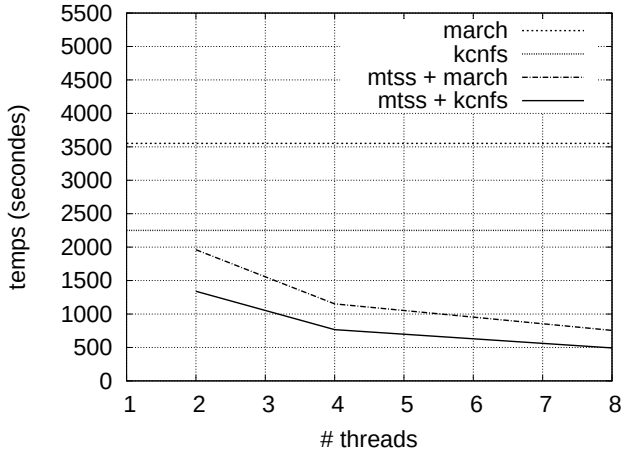


FIGURE 6.5 – Performances de solveurs type « aléatoire » parallélisés par MTSS

Les courbes d'efficacité ont été calculées en fonction des performances du solveur SAT parallélisé, ce sont donc des courbes d'efficacité relative pour les solveurs séquentiels. Compte tenu des performances de `kcnfs` sur ce type de formules, sa courbe d'efficacité est même la courbe d'efficacité absolue pour le problème 3-SAT aléatoire. Les performances sont plutôt bonnes car l'efficacité se situe aux alentours de 75 % sur 4 cœurs de calcul et de 60 % pour 8 cœurs. Ce sont des résultats intéressants, surtout si on garde à l'esprit que les codes sources de ces solveurs n'ont pas été modifiés. À l'image de MTSS

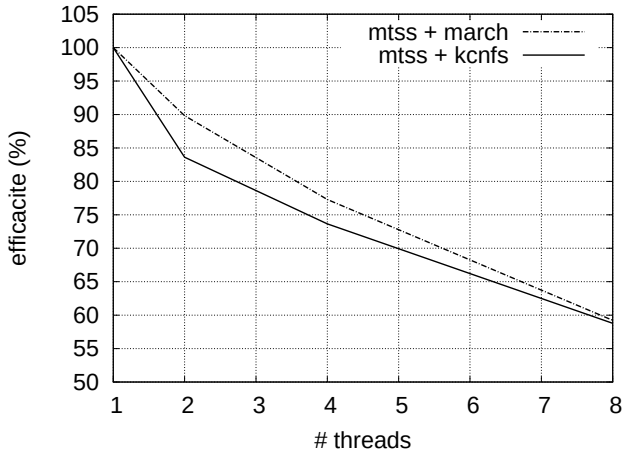


FIGURE 6.6 – Efficacité de solveurs type « aléatoire » parallélisés par MTSS

seul, ici les différents lancements ont des temps de calcul très proches. MTSS est encore une fois robuste, même en parallélisant des solveurs externes.

6.6.2 Solveurs industriels

Les formules testées pour ces expérimentations sont tirées de la SAT-Race 2008. Pour choisir les formules de ce test, nous avons posé une limite de 4h30min pour les exécutions séquentielles. Les formules trop longues (pas résolues avant la limite) à résoudre ou les trop petites (de l'ordre de quelques dizaines de secondes) ont été écartées car nous voulions un temps de référence significatif pour calculer les accélérations et l'efficacité de notre approche. Nous avons au final sélectionné 28 formules satisfaisables (SAT) et 28 formules insatisfaisables (UNSAT). Le solveur parallélisé était `Minisat2.0` [SE08], il l'a été sur 4, 8 et 16 cœurs. La limite de mutation a été fixée à $log_2(n) + 1$ avec n le nombre de *threads*.

Les courbes représentent la moyenne des 28 formules satisfaisables et celle des 28 formules insatisfaisables. Plusieurs exécutions ont été lancées pour chaque test, la moyenne ne reprend que les meilleurs temps de chaque couple nombre de *threads*, formule. Les courbes de temps de calcul et d'efficacité sont respectivement données par les figures 6.7 et 6.8. Les tableaux 6.1 et 6.2 montrent plus précisément le temps de calcul, le nombre de nœuds développés, l'accélération et la quantité de *travail* (terme expliqué plus bas) effectuée pour chaque famille de formules.

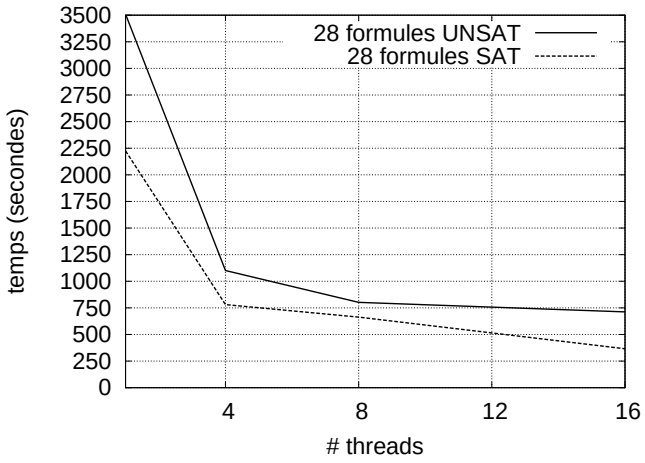


FIGURE 6.7 – Performances de `MiniSat` parallélisé par MTSS

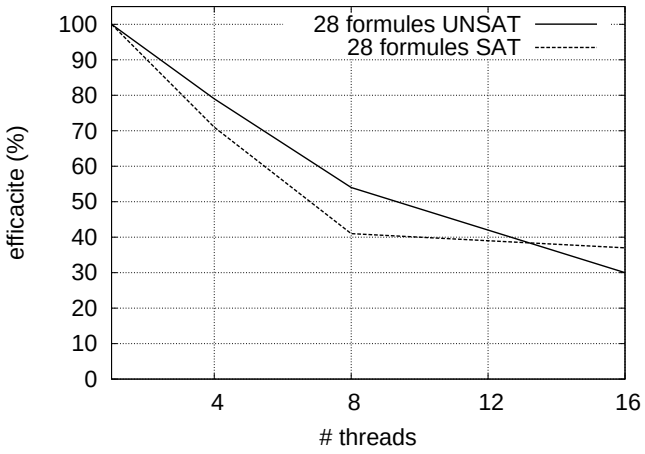


FIGURE 6.8 – Efficacité de `MiniSat` parallélisé par MTSS

TABLE 6.1 – Parallélisation de Minisat (résultats séquentiels et pour 4 *threads*)

			Minisat		4 *threads*			
famille/formule	valeur	# formules	temps (s.)	# nœuds	temps (s.)	# nœuds	acc.	travail
ibm-2002-*	SAT	8	17 134,52	9 485 784	6 659,43	23 553 521	5,54	2,36
ibm-2004-*	SAT	5	32 758,60	18 286 884	15 444,60	46 599 227	16,65	2,61
mizh-md5-*	SAT	5	42 735,25	36 164 552	20 103,10	72 333 595	2,87	1,76
mizh-sha0-*	SAT	5	68 711,13	61 082 802	24 574,62	95 819 568	9,24	1,11
palac-sn7-ipc5-h16.cnf	SAT	1	4 741,42	1 052 585	403,42	471 804	11,75	0,45
post-c32s-gcdm16-22.cnf	SAT	1	561,45	751 605	437,20	1 307 639	1,28	1,74
schup-l2s-motst-2-k315.cnf	SAT	1	183,87	112 514	220,64	111 200	0,83	0,99
simon-s02b-r4b1k1,2.cnf	SAT	1	714,05	1 628 748	134,38	1 465 152	5,31	0,90
simon-s03-w08-15.cnf	SAT	1	182,21	152 647	584,69	277 155	0,31	1,82
cmu-bmc-longmult*	UNSAT	2	75 165,73	65 182 330	26 383,05	99 873 414	2,54	1,14
een-pico-*	UNSAT	2	76 442,08	68 283 806	27 469,46	102 928 547	1,67	1,10
fuhs-aprove-*	UNSAT	2	155 023,30	136 140 768	55 688,42	205 352 879	1,99	1,04
goldb-heqc-*	UNSAT	2	246 685,28	226 008 537	86 190,76	317 550 475	7,84	0,89
ibm-2002-*	UNSAT	4	438 837,90	387 698 426	150 957,01	540 159 239	3,49	1,14
ibm-2004-29-k25.cnf	UNSAT	1	2 138,70	1 751 378	415,72	2 036 559	5,14	1,16
manol-pipe-*	UNSAT	7	29 029,23	46 191 581	5 962,05	48 508 567	5,94	1,03
post-*	UNSAT	4	66 926,59	105 084 902	14 440,39	100 657 566	2	1,20
schup-l2s-*	UNSAT	2	122 305,45	184 269 572	30 925,75	174 722 222	1,46	1,19
simon-*	UNSAT	2	216 901,29	325 594 704	58 479,09	308 891 177	0,88	1,27

TABLE 6.2 – Parallélisation de Minisat (résultats pour 8 et 16 *threads*)

			8 *threads*				16 *threads*			
famille/formule	valeur	# formules	temps (s.)	# nœuds	acc.	travail	temps (s.)	# nœuds	acc.	travail
ibm-2002-*	SAT	8	4 329,82	31 982 221	14,46	3,06	3 886,56	39 159 952	21,26	3,85
ibm-2004-*	SAT	5	9 636,57	62 148 272	17,44	3,31	6 678,85	72 770 871	26,54	3,63
mizh-md5-*	SAT	5	13 560,39	97 200 033	6,52	2,43	9 308,59	115 914 242	3,75	2,80
mizh-sha0-*	SAT	5	22 195,41	133 577 147	7,94	1,67	12 179,88	150 027 757	14,97	1,67
palac-sn7-ipc5-h16.cnf	SAT	1	118,67	358 681	39,95	0,34	411,79	1 793 397	11,51	1,70
post-c32s-gcdm16-22.cnf	SAT	1	215,31	1 576 504	2,61	2,10	337,22	1 678 120	1,66	2,23
schup-l2s-motst-2-k315.cnf	SAT	1	154,32	112 745	1,19	1,00	97,61	126 009	1,88	1,12
simon-s02b-r4b1k1,2.cnf	SAT	1	102,70	1 243 807	6,95	0,76	21,46	668 713	33,27	0,41
simon-s03-w08-15.cnf	SAT	1	198,37	193 072	0,92	1,26	151,02	167 012	1,21	1,09
cmu-bmc-longmult*	UNSAT	2	23 034,39	137 641 065	1,71	1,26	13 240,92	154 961 560	2,09	1,34
een-pico-*	UNSAT	2	23 697,55	140 837 141	1,42	1,22	13 685,15	158 483 191	1,39	1,42
fuhs-aprove-*	UNSAT	2	49 010,50	281 518 861	1,9	1,27	28 592,11	317 648 494	1,77	1,61
goldb-heqc-*	UNSAT	2	75 858,09	431 629 697	10,87	1,03	44 430,02	487 370 945	20,07	1,46
ibm-2002-*	UNSAT	4	131 406,86	733 461 316	4,94	1,42	77 492,85	827 080 980	6,01	1,64
ibm-2004-29-k25.cnf	UNSAT	1	307,84	2 328 117	6,95	1,33	177,65	2 579 366	12,04	1,47
manol-pipe-*	UNSAT	7	3 157,93	60 337 757	10,37	1,27	3 420,73	74 265 883	11,56	1,51
post-*	UNSAT	4	8 241,52	119 938 250	3,18	1,37	9 598,10	144 726 005	2,89	1,48
schup-l2s-*	UNSAT	2	19 628,96	208 211 324	1,67	1,32	21 402,34	249 115 894	2	1,36
simon-*	UNSAT	2	37 834,97	364 393 677	1,1	1,35	40 770,37	439 294 019	1,2	1,67

Il est notable que le temps de calcul est grandement réduit en utilisant 4 cœurs de calcul. Ensuite, la pente est plus douce. L'efficacité montre ce phénomène : 70% sur 4 cœurs pour les formules satisfaisables et 80% pour les formules insatisfaisables, ce sont de bonnes efficacités. En utilisant 8 cœurs, l'efficacité est moyenne (entre 40% et 55%), puis elle tombe à cause d'une stagnation du temps de calcul. Si cela est vrai pour la moyenne, il est cependant intéressant de voir que sur certaines formules, les performances sont très bonnes (voir les tableaux 6.1 et 6.2). L'accélération de `Minisat2.0` grâce à la parallélisation par MTSS est parfois super linéaire, l'efficacité est donc supérieure à 100 %. Comme montré en chapitre 3, cela peut survenir lorsque la formule est satisfaisable, et grâce à l'échange de clauses, même pour les formules insatisfaisables. Les échanges s'opèrent à chaque redémarrage. Le code source n'a pas été modifié en profondeur, seulement cet échange a été intégré à `Minisat2.0` ainsi qu'une fonction pour assimiler le chemin de guidage.

D'un côté, nous avons de bonnes accélérations, mais nous constatons aussi des performances très médiocres. Alors que les exécutions sur les formules aléatoires se révélaient très robustes, ici on observe des résultats très inégaux. Même entre plusieurs exécutions sur la même formule, il peut y avoir des différences importantes. Pour comprendre ces résultats, il faut analyser la colonne « travail ». Cette colonne est le multiplicateur du nombre de nœuds développés en séquentiel pour la version parallèle. On remarque que le travail est souvent supérieur à 1. Le nombre de nœuds développés en parallèle est donc souvent plus grand que le nombre de nœuds développés par `Minisat2.0` seul. Nous expliquons cela par le fait que les solveurs industriels, de par leur mécanisme, ne se comportent pas bien lorsque certaines variables sont fixées : celles du chemin de guidage sont fixes même après un redémarrage du solveur externe. De plus, l'heuristique naïve que nous avons mise en place pour les formules industrielles, MOMS, est rapide mais n'offre pas de bonnes performances. Pour l'instant, le principal frein pour que MTSS ait de très bonnes performances de manière plus systématique est dû au nombre de nœuds développés. Le travail fourni par les exécutions parallèles sont en général trop important. Un autre facteur s'ajoute aux performances non stables : l'échange des clauses ne se fait pas de manière déterministe. Prenons deux sous-arbres A et B résolus par des exécutions différentes du solveur externe. Il est possible que les clauses apprises dans B ne servent pas A mais que l'inverse soit vrai. Or, rien dans les échanges de clauses ne force l'ordre des échanges, même entre deux exécutions successives sur la même formule.

Beaucoup de paramètres influent sur les performances, en particulier tout ce qui concerne les clauses échangées (politiques de réduction ou d'échange, taille limite des clauses ou de la base, ...). La limite de mutation influe aussi pour une part importante dans les performances, de même que l'heuristique de branchement dans la partie MTSS en haut de l'arbre. Il existe donc beaucoup de pistes pour améliorer cette partie de MTSS. Par exemple, MTSS ne gèrera pas la subdivision en profondeur du dernier sous-arbre calculé par le solveur externe, même si ce dernier sous-arbre est très long à calculer.

6.7 Conclusion

D'après les expérimentations menées, il est intéressant de voir que les concepts choisis et la façon de les mettre en œuvre ont amené MTSS à être un solveur parallèle très efficace sur ses formules de prédilection, à savoir les formules aléatoires. En effet, 80% d'efficacité sur 8 cœurs de calcul, tout en assurant que différentes exécutions seront quasiment similaires en temps de calcul, cela fait de MTSS un solveur fiable et robuste. Il est évident que la gestion mémoire spécifique ainsi que les concepts d'arbre de guidage et de processus riche / pauvres sont des choix cohérents sur architecture CC-UMA où les multi-cœurs sont aujourd'hui très répandus. Nous avons mené nos recherches dans ce sens et ça a fonctionné. En ce qui concerne l'outil de parallélisation de solveurs externes implémenté dans MTSS, il est très intéressant de voir le comportement stable qu'il a sur des formules aléatoires. De plus, il offre des performances plutôt bonnes avec une efficacité d'environ 60% sur 8 cœurs de calcul. Pour les formules industrielles, les performances sont moins uniformes, il y a du très bon et du moins bon. Les bonnes performances s'expliquent par les gains qu'offrent le parallélisme, mais les mauvaises peuvent se comprendre par des choix stratégiques pour l'instant basiques, en particulier l'heuristique de branchement en haut de l'arbre.

Conclusion et perspectives

Conclusion

Le défi qui se présente à tous les domaines de recherche dont l'objectif est l'efficacité des algorithmes est de passer d'une cinquantaine d'années de développement séquentiel à une approche parallèle. Comme nous l'avons vu, même pour un problème très irrégulier tel que SAT, le parallélisme n'est pas qu'un simple gain de vitesse, les accélérations, dans certains cas, peuvent être super linéaires. Les problèmes $\in \mathcal{NP}$ sont généralement étudiés dans un cadre séquentiel, c'est se priver des apports que peut offrir la programmation parallèle, en particulier pour les problèmes de décision tels que SAT. Aujourd'hui, et même depuis quelques années maintenant, les processeurs multi-cœurs sont la norme, il est grand temps de profiter de cette technologie devenue bon marché et accessible. Elle permet de mettre en place une approche très collaborative, qui n'avait jusqu'à présent jamais été imaginée, et encore moins mise en œuvre. Avec MTSS, nous avons amené ce type d'approche très axée sur la collaboration, permettant d'imaginer de multiples applications. Le principe riche / pauvres peut être repris pour d'autres champs de recherche souhaitant aborder le parallélisme : un processus riche, proche d'un algorithme séquentiel travaille sur le problème, tandis que des processus pauvres calculent quelques informations en avance du riche dans son processus de résolution. Pour le problème SAT, nous avons couplé ce principe fondamental à celui d'arbre de guidage pour distribuer et équilibrer la charge. Tandis que sur le chemin du riche, les pauvres recalculent les sous-formules sur lesquelles travailler, entre eux, ils se mettent à disposition leurs résultats intermédiaires afin de reprendre le calcul où que ce soit dans l'arbre, en un temps constant. Ce calcul intermédiaire peut être repris par n'importe quel processus (riche ou pauvre) de MTSS. Mais le plus important dans le concept d'arbre de guidage est que nous avons amené un grain de parallélisme très fin à la résolution de SAT en parallélisant complètement en mémoire le développement d'un arbre binaire de recherche. Ce grain est de l'ordre d'un nœud dans l'arbre de guidage alors que la plupart des autres solveurs implémentent un gros grain de parallélisme en travaillant au niveau des sous-arbres. Cette approche permet de diversifier les processus pauvres à une multitude de traitements locaux aux nœuds. Ces traitements ne manquent pas dans la littérature SAT, ce qui présage de nombreuses améliorations possibles à MTSS.

Malgré une gestion très fine du grain de parallélisme, nous avons montré que d'excellentes accélérations étaient possibles sur des machines CC-UMA (par exemple les multi-cœurs). C'était l'objectif de cette thèse. Toutefois, nous avons dû mener un travail

particulier sur la gestion de la mémoire et un travail assez minutieux sur les synchronisations des processus afin que les défauts de cache ne détériorent pas les performances et pour minimiser les attentes entre les processus. Ce travail, autant au niveau théorique que pratique a abouti à un solveur — MTSS— efficace à plus de 80% sur 8 cœurs de calcul sur des formules générées aléatoirement de 500 variables et toutes insatisfaisables, autrement dit, 80% est la pire efficacité que nous pouvons relevé sur 8 cœurs de calcul pour des formules de 500 variables. Nous avons mis en place le minimum de tâches nécessaires pour que les pauvres puissent développer l'arbre de guidage tous ensemble. Mais, grâce à la définition de MTSS qui permet une multitude d'actions, et à l'arbre de guidage qui permet un travail à grain fin mais aussi à gros grain, nous avons aussi implémenté la tâche la plus macroscopique possible, celle ayant le plus gros grain parallèle, en offrant l'opportunité de paralléliser d'autres solveurs SAT. Ce choix est motivé par le fait que les solveurs efficaces maintenus par la communauté de chercheurs en SAT aujourd'hui sont majoritairement séquentiels. Nous espérons que MTSS puisse servir à d'autres chercheurs pour observer le comportement de leurs solveurs en parallèle, ou tout simplement à résoudre des formules trop complexes pour être résolues par un seul processeur. De très bons résultats ont été observés dans la parallélisation de solveurs destinés à résoudre des formules aléatoires : environ 60% d'efficacité sur 8 cœurs de calcul. De plus ces résultats étaient constants entre plusieurs exécutions et n'ont demandé aucune modification du code source de ces solveurs. La parallélisation d'un solveur industriel a montré des performances moins constantes, et en moyenne moins bonnes mais de grandes accélérations ont tout de même été relevées sur certains tests, voire parfois des accélérations super linéaires. Par exemple une accélération de 39,95 a été relevée sur 8 cœurs (efficacité de 500%) sur une instance satisfaisable et une accélération de 20,07 a été mesurée sur une famille de *benchmarks* insatisfaisables sur 16 cœurs de calcul (efficacité de 125% sur des formules n'admettant pas de solution).

Le parallélisme de SAT par MTSS amène donc une bonne robustesse, de bonnes performances et de bonnes accélérations pour un solveur complet destiné à résoudre les formules générées aléatoirement. Ce type de solveur n'existait pas encore dans un cadre multi-cœurs. MTSS parallélise des solveurs SAT existants, permettant d'obtenir des gains super linéaires sur les formules satisfaisables et même sur les formules insatisfaisables.

Perspectives

Le plus intéressant dans MTSS est le nombre de ses perspectives possibles. En effet, nous envisageons trois thèmes à aborder à plus ou moins long terme.

Tout d'abord, à court terme, il nous parait indispensable d'améliorer la partie concernant la parallélisation de solveurs externes, en particulier pour l'échange des clauses et l'heuristique de branchement utilisée dans le cas des formules industrielles. Nous devons mettre en place différentes stratégies pour connaître l'impact du haut de l'arbre, cette partie que nous forçons pour les solveurs industriels et qui semble être un frein à leurs performances. Ensuite, la base de clauses doit subir une amélioration dans sa gestion, en particulier dans son compactage et dans ses critères d'échange. Enfin, il

serait utile de développer une politique d'équilibrage des derniers sous-arbres difficiles à résoudre par le solveur externe. Le but de ces travaux est de fournir un meilleur outil de parallélisation pour les solveurs industriels, en particulier des performances plus homogènes, plus robustes.

À moyen terme, il serait intéressant d'enrichir les processus pauvres. Pour le moment, ils développent l'arbre de guidage ou calculent tout un sous-arbre grâce à un solveur externe, mais la définition de ces processus permet beaucoup plus. Il est envisageable d'intégrer à MTSS n'importe quel traitement local, pré-traitement, apprentissage ... de l'état de l'art de SAT en tant que processus pauvre. Il est également tout à fait possible d'implémenter une méthode de résolution incomplète en tant que fonction de pauvre. C'est là la force de la définition riche / pauvre : ironiquement le processus pauvre est riche de perspectives. Tout ce qui est imaginable en tant que processus pauvre interne à MTSS l'est aussi de manière externe, à l'image de la parallélisation de solveurs externes déjà implémentée. Toutefois, il faut conserver à l'esprit que la création d'un nombre très grand de processus au sein d'un système informatique a des conséquences sur ses performances. Lancer un traitement externe et donc un nouveau processus sur des sous-arbres ou sur certains nœuds est certainement plus bénéfique que de le faire à chaque nœud de l'arbre.

À long terme, nous voulons faire évoluer MTSS, qui est pour l'instant un bon solveur pour architectures CC-UMA en solveur parallèle déployable sur n'importe quel super calculateur moderne. Ces derniers sont généralement de type NoRMA avec des nœuds de type CC-UMA ou CC-NUMA. MTSS est efficace sur chaque nœud CC-UMA ou dans une moindre mesure sur les nœuds CC-NUMA mais pour être un solveur réellement extensible sur les machines actuelles, il est nécessaire de développer une couche DRAM (en utilisant par exemple MPI pour transférer les messages). Ce solveur pourrait reprendre le principe de chemin de guidage idéal pour ce type de programmation et le classique modèle maître / esclave. Chaque esclave de ce nouveau solveur serait constitué d'un riche et de plusieurs pauvres dont le nombre dépendrait du nombre de cœurs de calcul au niveau du nœud sur lequel il s'exécute. Nous gardons aussi à l'esprit l'actuel intérêt que peuvent représenter le calcul sur carte graphique (*GPGPU : General-Purpose Processing on Graphics Processing Units*), par exemple en utilisant un solveur SAT pour GPU en tant que solveur externe, ou en implémentant une version de MTSS avec sa vision très collaborative sur GPU.

Si ces trois champs d'étude sont développés, MTSS représenterait un solveur parallèle très polyvalent et performant pour résoudre SAT. Il est déjà très efficace, et cela serait renforcé, il pourrait être exécuté sur tout type de machine parallèle, et de plus, il serait possible de paralléliser de manière efficace n'importe quel type de solveur sur n'importe quel super calculateur. Cette perspective est intéressante et motivante.

Bibliographie

[AS09] Gilles Audemard and Laurent Simon. Glucose : a solver that predicts learnt clauses quality. Technical report, Solver Description, SAT competition 2009, 2009.

[BBC+01] Béla Bollobás, Christian Borgs, Jennifer T. Chayes, Jeong Han Kim, and David Bruce Wilson. The scaling window of the 2-sat transition. *Random Struct. Algorithms*, 18(3) :201–256, 2001.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99 : Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[Bie09] Armin Biere. P{re,i}cosat@sc'09. Technical report, Solver Description, SAT competition 2009, 2009.

[BMZ05] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation : An algorithm for satisfiability. *Random Struct. Algorithms*, 27(2) :201–226, 2005.

[BS94] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. In *Third International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, 1994.

[BSK03] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7) :969–994, 2003.

[CA96] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-sat. *Artificial Intelligence*, 81(1-2) :31–57, 1996.

[CF91] Michel Cosnard and Afonso Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1 :103–111, 1991.

[Coo71] 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.

[CR92] Vasek Chvátal and B. Reed. Mick gets some (the odds are on his side). In *33rd Annual Symposium on Foundations of Computer Science*, pages 620–627. IEEE, 1992.

[CS08] Geoffrey Chu and Peter J. Stuckey. Pminisat : A parallelization of minisat 2.0. Technical report, Solver Description, SAT-Race 2008, 2008.

[CW06] Wahid Chrabakh and Richard Wolski. Gridsat : Design and implementation of a computational grid application. *J. Grid Comput.*, 4(2) :177–193, 2006.

[Dar08] Sylvain Darras. *Traitements locaux dans les arbres de recherche pour SAT et Max-SAT*. PhD thesis, Université de Picardie Jules Verne, 2008.

[DC91] Olivier Dubois and Jacques Carlier. Probabilistic approach to the satisfiability problem. *Theor. Comput. Sci.*, 81(1) :65–75, 1991.

[DD03] Gilles Dequen and Olivier Dubois. kcnfs : An efficient solver for random k-sat formulae. In *SAT*, pages 486–501, 2003.

[DD06] Gilles Dequen and Olivier Dubois. An efficient approach to solving random sat problems. *J. Autom. Reasoning*, 37(4) :261–276, 2006.

[DDD+05] Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, and Lakhdar Saïs. Using boolean constraint propagation for sub-clauses deduction. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 757–761. Springer, 2005.

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

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

[Dub91] Olivier Dubois. Counting the number of solutions for instances of satisfiability. *Theor. Comput. Sci.*, 81(1) :49–64, 1991.

[Dub01] Olivier Dubois. Upper bounds on the satisfiability threshold. *Theor. Comput. Sci.*, 265(1-2) :187–197, 2001.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, 12(5) :662–675, 1986.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FDH04] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver : Design and implementation, 2004.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *IEEE*, 54(12) :1901–1909, December 1966.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, 1972.

[FR96] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1) :67–70, 1996.

[Fri99] Ehud Friedgut. Sharp thresholds of graph properties, and the k-sat problem. *J. Amer. Math. Soc*, 12 :1017–1054, 1999.

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. clasp : A conflict-driven answer set solver. Technical report, Solver Description, SAT competition 2009, 2009.

[GMPW95] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. Scaling effects in the csp phase transition. In *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, 1995, Proceedings*, volume 976 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 1995.

[Goe92] Andreas Goerdt. A threshold for unsatisfiability. In *Mathematical Foundations of Computer Science 1992, 17th International Symposium, MFCS'92, Proceedings*, volume 629 of *Lecture Notes in Computer Science*, pages 264–274. Springer, 1992.

[Gol79] A. Goldberg. On the complexity of the satisfiability problem. Technical Report 16, Courant Computer Science Report, New York University, 1979.

[GW93] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI)*, pages 28–33, 1993.

[GW94a] Ian P. Gent and Toby Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70(1-2) :335–345, 1994.

[GW94b] Ian P. Gent and Toby Walsh. The hardest random sat problems. In *KI-94 : Advances in Artificial Intelligence, 18th Annual German Conference on Artificial Intelligence, 1994, Proceedings*, volume 861 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 1994.

[GW94c] Ian P. Gent and Toby Walsh. The sat phase transition. In *Proceedings of 11th European Conference on Artificial Intelligence (ECAI)*, pages 105–109, 1994.

[GW96a] Ian P. Gent and Toby Walsh. Phase transitions and annealed theories : Number partitioning as a case study. In *12th European Conference on Artificial Intelligence, 1996, Proceedings*, pages 170–174. John Wiley and Sons, Chichester, 1996.

[GW96b] Ian P. Gent and Toby Walsh. The tsp phase transition. *Artificial Intelligence*, 88 :105–109, 1996.

[HHHLB06] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2006.

[HJS08] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat : a multicore sat solver. Technical report, Solver Description, SAT-Race 2008, 2008.

[HJS09a] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 499–504, Pasadena, California, USA, july 2009. Morgan Kaufmann.

[HJS09b] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat : a parallel sat solver. *JSAT*, 6 :245–262, 2009.

[HK05] Edward A. Hirsch and Arist Kojevnikov. Unitwalk : A new sat solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1-4) :91–111, 2005.

[HvM06] Marijn Heule and Hans van Maaren. March_dl : Adding adaptive heuristics and a new branching strategy. *JSAT*, 2(1-4) :47–59, 2006.

[JK07] C. Jaillet and M. Krajecki. Parallel programming with openmp : a new memory allocation model avoiding cache faults. In *International Workshop on OpenMP 2007 (IWOMP2007)*, volume 4935 of *Lecture Notes in Computer Science*, pages 148–152, Tsinghua University, Beijing, China, june 2007. Springer.

[JLU01] B. Jurkowiak, C. M. Li, and G. Utard. Parallelizing Satz Using Dynamic Workload Balancing. In *Proc. of Workshop on Theory and Application of Satisfiability Testing (Sat'2001)*, pages 205–211, 2001.

[Joh88] Eric E. Johnson. Completing an mimd multiprocessor taxonomy. *SIGARCH Comput. Archit. News*, 16(3) :44–47, 1988.

[JS97] Roberto J. Bayardo Jr. and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1 :167–187, 1990.

[KB03] Sven Karlsson and Mats Brorsson. Priority based messaging for software distributed shared memory. *Cluster Computing*, 6(2) :161–169, 2003.

[Kok98] Dan Kokotov. Psolver : Distributed sat solver framework. Technical report, available on WebSite : http ://sdg.csail.mit.edu/satsolvers/PSolver, 1998.

[KS98] Henry A. Kautz and Bart Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *International Conference on*

Automated Planning and Scheduling (ICAPS) / Conference on Artificial Intelligence Planning Systems (AIPS), pages 181–189, 1998.

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

[Lan05] Martin Lange. Solving parity games by a reduction to sat. In *Proc. of the Workshop on Games in Design and Verification, GDV'05*, 2005.

[Lav09] Ivan Lavallée. *Complexité et algorithmique avancée (une introduction).* Éditions Hermann, Paris, 2 edition, february 2009.

[LH05] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2005.

[Li00] Chu Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.

[LM82] Miron Livny and Myron Melman. Load balancing in homogeneous broadcast distributed systems. In *SIGMETRICS*, pages 47–56, 1982.

[LSB04] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Early conflict detection based bcp for sat solving. In *SAT*, 2004.

[LSB05] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Speedup techniques utilized in modern sat solvers. In *SAT*, pages 437–443, 2005.

[LSB07] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *ASP-DAC '07 : Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[LSH06] Frédéric Lardeux, Frédéric Saubion, and Jin-Kao Hao. Gasat : A genetic local search algorithm for the satisfiability problem. *Evolutionary Computation*, 14(2) :223–253, 2006.

[LWZ07] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2007.

[LZ07] W. Wei C. M. Li and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for sat. Technical report, Solver Description, SAT competition 2007, 2007.

[MFM04] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004 : An efficient sat solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[Moh09] Nouredine Ould Mohamedou. *L'exploitation des structures dans les problèmes SAT et Max-SAT.* PhD thesis, Université de Picardie Jules Verne, 2009.

[MSG97] Bertrand Mazure, Lakhdar Saïs, and Eric Grégoire. Tabu search for sat. In *Proceedings of the 14th American National Conference on Artificial Intelligence (AAAI)*, pages 281–285, 1997.

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of sat problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*, pages 459–465, 1992.

[MZ06] Panagiotis Manolios and Yimin Zhang. Implementing survey propagation on graphics processing units. In *SAT*, pages 311–324, 2006.

[MZK+99] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. 2+p-sat : Relation of typical-case complexity to the nature of the phase transition. *Random Struct. Algorithms*, 15(3-4) :414–435, 1999.

[NDS+04] Eugene Nudelman, Alex Devkar, Yoav Shoham, Kevin Leyton-Brown, and Holger Hoos. Satzilla : An algorithm portfolio for sat, 2004.

[OSM02] R. Ostrowski, L. Saïs, and B. Mazure. Lsat solver. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, 2002.

[PG07] D. N. Pham and C. Gretton. gnovelty+. Technical report, Solver Description, SAT competition 2007, 2007.

[PG09] Duc Nghia Pham and Charles Gretton. gnovelty+ (v. 2). Technical report, Solver Description, SAT competition 2009, 2009.

[PIM06] Allon Percus, Gabriel Istrate, and Cristopher Moore. *Computational Complexity and Statistical Physics (Santa Fe Institute Studies in the Sciences of Complexity Proceedings).* Oxford University Press, Inc., New York, NY, USA, 2006.

[PRR+07] Nachiketh R. Potlapally, Anand Raghunathan, Srivaths Ravi, Niraj K. Jha, and Ruby B. Lee. Aiding side-channel attacks on cryptographic software with satisfiability-based analysis. *IEEE Trans. VLSI Syst.*, 15(4) :465–470, 2007.

[PTGS08] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *JSAT*, 4(2-4) :149–172, 2008.

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

[SB06] Christine Solnon and Derek Bridge. An ant colony optimization metaheuristic for subset selection problems. *System Engineering using Particle Swarm Optimization*, pages 7–29, 2006.

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. Pasat - parallel sat-checking with lemma exchange : Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9 :205–216, 2001.

[SE08] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0 - sat race 2008 editions. Technical report, SAT-Race 2008 : Solver Descriptions, 2008.

[SKC96] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, 1996.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*, pages 440–446, 1992.

[SM07] D. Singer and A. Monnet. JaCk-SAT : A New Parallel Scheme to Solve the Satisfiability Problem (SAT) based on Join-and-Check. In *Proc of Parallel Processing and Applied Mathematics*, Gdansk, 2007.

[SMV87] Ewald Speckenmeyer, Burkhard Monien, and Oliver Vornberger. Superlinear speedup for parallel backtracking. In *ICS*, pages 985–993, 1987.

[Spe93] W. Spears. Simulated annealing for hard satisfiability problems. *Technical report, Naval Research Laboratory, Washington D.C.*, 1993.

[Spe08] Ivor Spence. tts : A sat-solver for small, difficult instances. *JSAT*, 4(2-4) :173–190, 2008.

[Spe09] Ivor Spence. Ternary tree solver (tts-5-0). Technical report, Solver Description, SAT competition 2009, 2009.

[SS96] Joao P. Marques Silva and Karem A. Sakallah. Grasp a new search algorithm for satisfiability. In *ICCAD '96 : Proc. of the 1996 IEEE/ACM Intern. Conf. on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In *Formal Methods in System Design*, pages 82–99. Springer-Verlag, 1998.

[TH04] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat : An implementation and experimentation environment for sls algorithms for sat & max-sat. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, Online Proceedings*, 2004.

[TOU09] Kota Tsuyuzaki, Kei Ohmura, and Kazunori Ueda. satake : solver description. Technical report, Solver Description, SAT competition 2009, 2009.

[TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2) :254–272, June 2009.

[VSDK08] Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. On multi-threaded satisfiability solving with openmp. In *IWOMP*, pages 146–157, 2008.

[VSDK09a] P. Vander-Swalmen, G. Dequen, and M. Krajecki. A collaborative approach for multi-threaded sat solving. *International Journal of Parallel Programming*, 37(3) :324–342, 2009.

[VSDK09b] Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. Automatic parallel sat solving using mtss. In *HPCS*, 2009.

[VSDK09c] Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. Toward easy parallel sat solving. In *21st International Conference on Tools with Artificial Intelligence*, Newark (NYC Metropolitan Area), New Jersey, USA, November 2009.

[WL09] Wanxia Wei and Chu Min Li. Switching between two adaptive noise mechanisms in local search for sat. Technical report, Solver Description, SAT competition 2009, 2009.

[WM85] Yung-Terng Wang and Robert J. T. Morris. Load sharing in distributed systems. *IEEE Trans. Computers*, 34(3) :204–217, 1985.

[XHHLB09] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla2009 : an automatic algorithm portfolio for sat. Technical report, Solver Description, SAT competition 2009, 2009.

[ZB94] Hantao Zhang and Maria Paola Bonacina. Cumulating search in a distributed computing environment : A case study in parallel satisfiability. In *Proc. of the First Int. Symp. on Parallel Symbolic Computation*, pages 422–431. Publishing Company, 1994.

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

[Zha93] Hantao Zhang. Sato : A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22 :1–3, 1993.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 279–285, 2001.

Table des figures

Table des algorithmes

Annexe

Guide d'utilisation de MTSS

Vous trouverez MTSS ainsi que la bibliothèque pour paralléliser facilement vos implémentations de solveurs SAT (MTSS.h) sur le site Internet hébergé par l'URCA à l'adresse `http://www.parallel-sat.net`. Les options disponibles pour MTSS sont décrites ci-dessous.

Options

- '-p n' : avec n le nombre de *threads* pauvres à lancer.
- '-t z' : à partir du seuil z (en % du nombre de variables), les pauvres ne travailleront pas (par défaut, z=5%)
- '-fm' : Force l'utilisation mémoire (le comportement de base préfère allouer moins de nœuds pour l'arbre de guidage que le nombre de nœuds par défaut si la mémoire ne le permet pas mais le nombre de nœuds développables en parallèle par les pauvres en sera réduit ; cette option passe outre). À savoir que sous un système GNU/Linux, la taille mémoire libre est généralement faible car la RAM est utilisée comme une mémoire cache du disque dur. Vous pouvez activer cette option sans problème, toutefois, si vous constatez des performances très faibles, alors retirez cette option.
- '-f *[bsh_3sat, bsh_ksat, moms]*' : Force l'utilisation d'une heuristique en particulier. Choix entre 3 heuristiques.
- '-rtf' : « remove temp files », efface les fichiers temporaires créés pendant l'exécution de MTSS avec l'option 'es' ou 'eh' activée(s).
- '-es *config_file*' : les pauvres et le riche exécuteront un solveur externe en fonction des données passées par le fichier de configuration.
- '-eh *config_file*' : les pauvres et le riche exécuteront une heuristique externe en fonction des données passées par le fichier de configuration.
- '-h' ou '–help' : affiche une aide en ligne et en anglais équivalente à celle-ci

Comment rendre son solveur parallèle ?

Généralités

Utiliser l'option '-es' (pour *external solver*) et fournir l'emplacement d'un fichier de configuration qui devra suivre un certain formalisme.

- ligne 1 : un entier pour la limite de mutation qui correspond à la profondeur dans l'arbre de recherche.
- ligne 2 : un mot-clé expliquant le comportement que doit adopter MTSS vis-à-vis du solveur externe. Ce mot-clé indique à MTSS si il doit prendre en charge un échange de clauses ou pas. Le mots-clé reconnu par MTSS pour lui indiquer qu'un échange de clauses est demandé est : 'share_clauses'. N'importe quelle autre suite de caractères ici signifiera que MTSS ne partagera pas de clauses. Attention, cette ligne ne doit pas être vide.
- ligne 3 : chemin vers le binaire (exécutable) du solveur externe à paralléliser (500 caractères max)
- ligne 4 : nom du solveur (50 caractères maximum)
- lignes supplémentaires : arguments à passer au solveur externe. Un seul argument par ligne, et 50 caractères maximum par ligne. MTSS ne peut lire qu'un maximum de 5 arguments.

Le solveur externe doit pouvoir lire les fichiers au format DIMACS et doit afficher sa réponse au format DIMACS. Le format DIMACS est défini comme suit : une ligne de commentaire doit commencer par 'c', la première ligne du fichier lu est de la forme 'p cnf n m' avec n le nombre de variables et m le nombre de clauses. Ensuite, les clauses sont décrites comme une suite d'entiers signés terminées par un zéro pour signaler la fin de clause. L'affichage de la réponse commence par 's' puis le mot SAT ou UNSAT. L'affichage d'une solution, si le solveur le fait, commence par un 'v' puis il affiche les littéraux satisfaits par l'affectation des variables.

Dans le cas de solveurs destinés à résoudre les formules aléatoires, il n'est pas nécessaire de le modifier. Pour tirer un maximum de performances de la parallélisation des solveurs destinés à résoudre les formules industrielles, il est intéressant de partager les clauses apprises entre les exécutions du solveur externe.

Partage de clauses

Afin de donner la possibilité de partager les clauses apprises à l'ensemble MTSS et solveur externe, il est nécessaire de modifier le code source du solveur externe. Le principe de base réside dans deux étapes primordiales à ajouter :

1. Intégration du chemin de guidage : après avoir lu le fichier d'entrée, et avant de débuter le calcul, il est nécessaire de récupérer le chemin de guidage depuis MTSS. Pour cela il suffit d'appeler la fonction idoine de la bibliothèque MTSS.h (*mtss_share_receive_guiding_path*). Il sera nécessaire de sauvegarder le chemin reçu (à l'aide de *mtss_share_extract_guiding_path*) et de toujours commencer la simplification de la formule, à chaque redémarrage, par les littéraux de ce chemin.

2. Partage des clauses : avant chaque redémarrage du solveur, appeler une fonction qui se chargera d'envoyer à MTSS les nouvelles clauses apprises depuis le dernier redémarrage (*mtss_share_send_clauses*) puis récupérera celles apprises par les autres solveurs depuis le dernier partage (*mtss_share_receive_clauses*).

La bibliothèque MTSS.h fournit un ensemble de fonctions afin de faciliter la gestion des échanges entre MTSS et le solveur externe, mais un certain nombre de points sont à gérer par le ou les personnes en charge du solveur externe ; les voici :

- Puisque chaque solveur possède sa propre manière de coder les variables propagées ou les clauses, la librairie fournie fait le choix de ne gérer que des entiers signés (à l'instar d'un fichier au format DIMACS). Le solveur externe devra donc relire les informations reçues ou traiter celles à envoyer en fonction de cela.
- Afin de ne pas envoyer plusieurs fois les mêmes clauses à MTSS, le solveur externe devra gérer lui-même la sauvegarde de l'index (dans la base des clauses apprises) de la dernière clause partagée, et penser à le mettre à jour après toute modification de la taille de la base des clauses apprises.
- Le fait d'activer ou non le partage avec MTSS est laissé au(x) gestionnaire(s) du solveur externe. Par exemple en utilisant une variable de compilation ou par l'utilisation d'une option sur la ligne de commande. Ne pas oublier de conditionner les appels aux fonctions de la librairie car l'attente de communication avec MTSS est bloquant pour le processus (lecture dans un tube de communication).
- L'envoi de la réponse (SAT ou UNSAT) à MTSS s'effectue aussi par l'intermédiaire du tube de communication mis en place pour le partage de clauses. Par ce biais, il est également possible de renseigner MTSS du nombre de décisions prises pour l'obtention de la réponse. Pensez que si MTSS trouve ou reçoit une solution, alors il tuera tous les processus de solveurs externes en cours d'exécution. Dans un tel cas, si vous souhaitez tout de même informer MTSS du nombre de décisions prises, il vous faudra mettre en place un *handler* pour le signal SIG_INT (interruption).

Récapitulatif des fonctions utiles pour les solveurs externes :

- mtss_share_t : type de la structure à déclarer afin de gérer l'échange de clauses.
- void mtss_share_init(mtss_share_t *, int) : initialisation de la structure d'échange (à appeler une seule fois après la déclaration). L'entier en paramètre sert à forcer la taille du *buffer* d'échange, pour utiliser la valeur par défaut, il suffit de mettre la constante MTSS_UNDEF.
- void mtss_share_reset(mtss_share_t *) : remise à zéro de la structure d'échange (à utiliser entre une réception et un envoi, avant d'alimenter la structure pour l'envoi).

- int mtss_share_receive_guiding_path(mtss_share_t *) : Cette fonction attend le chemin de guidage donné par MTSS et retourne le nombre de décisions contenues

dans ce chemin.
- int mtss_share_extract_guiding_path(mtss_share_t *, int) : retourne une à une les décisions envoyées par le chemin de guidage. À appeler autant de fois que nécessaire.

- int mtss_share_add_literal(mtss_share_t *, int) : ajoute un littéral dans la structure dans la clause en cours de construction. Cette fonction et la suivante sont à utiliser si on préfère alimenter un à un chaque littéral (par exemple si les littéraux des clauses ne sont pas enregistrés sous la forme d'un entier signé).
- int mtss_share_end_clause(mtss_share_t *) : termine la clause en cours de construction pour passer à la suivante.
- int mtss_share_add_clause(mtss_share_t *, int *, int) : ajoute directement une clause sous la forme d'entiers signés dans la structure d'échange. L'entier en paramètre permet de donner la taille de la clause, mais il est possible de finir la clause par un zéro, et la fonction comprendra qu'elle a atteint la fin de la clause. Pour avoir ce comportement, il suffit de renseigner MTSS_UNDEF au lieu d'une taille.
- int mtss_share_send_clauses(mtss_share_t *) : Après avoir intégré les clauses à envoyer dans la structure d'échange, l'appel à cette fonction effectuera l'envoi à MTSS de ces clauses.

- int mtss_share_receive_clauses(mtss_share_t *) : Cette fonction attend des clauses depuis MTSS et retourne le nombre de clauses reçues.
- int mtss_share_get_clause_size(mtss_share_t *, int) : retourne la taille de la clause demandée.
- int mtss_share_get_clause_start(mtss_share_t *, int) : retourne l'indice de début de clause pour la clause demandée. Utile pour lire soi-même une clause depuis la structure si par exemple on ne code pas en interne les clauses sous forme d'entiers signés.
- int mtss_share_get_literal(mtss_share_t *, int, int) : retourne le littéral pointé par les entiers passés en paramètres depuis la structure d'échange (à l'image d'une matrice).
- int mtss_share_get_clause(mtss_share_t *, int, int *) : si les clauses sont enregistrées sous forme d'entiers signés, il est possible d'utiliser directement cette fonction pour récupérer le contenu d'une clause. Attention, le tableau d'entiers passé en paramètre doit pouvoir contenir la clause. Si vous souhaitez ajouter un zéro à la fin de cette clause, prévoyez un espace supplémentaire car cette fonction ne le fera pas.

- int mtss_share_send_result(mtss_share_t *, const int, int) : cette fonction enverra à MTSS le résultat (MTSS_RESULT_SAT ou MTSS_RESULT_UNSAT ou bien encore MTSS_RESULT_UNDEF en premier entier) ainsi que le nombre de décisions prises (le second entier) pendant la résolution.