Résolution parallèle de SAT: mieux collaborer pour aller plus loin

Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette

▶ **To cite this version:**

Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, Cédric Piette. Résolution parallèle de SAT: mieux collaborer pour aller plus loin. JFPC 2012, 2012, Toulouse, France. hal-00818239

HAL Id: hal-00818239
https://inria.hal.science/hal-00818239

Submitted on 26 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Résolution parallèle de SAT : mieux collaborer pour aller plus loin

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez et Cédric Piette *

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens

{audemard,hoessen,jabbour,lagniez,piette}@cril.fr

Résumé

La gestion de la base de clauses apprises est connue pour être une tâche ardue au sein des solveurs SAT. Dans le cadre des solveurs parallèles de type *portfolio*, la collaboration réalisée entre les différents processus au moyen de l'échange de clauses rend cette gestion plus complexe encore. Différentes techniques ont été proposées ces dernières années, mais les résultats pratiques restent en faveur de solveurs ayant une collaboration limitée voire nulle. Ceci est principalement dû au fait que chacun des processus doit gérer le grand nombre de clauses générées par les autres. Dans cet article, nous proposons une nouvelle technique efficace pour l'échange de clauses au sein d'un solveur parallèle. Contrairement aux méthodes actuelles, notre approche repose sur des politiques pour l'exportation et pour l'importation de clauses, tout en utilisant des techniques récentes, qui ont prouvées leur efficacité dans le cadre séquentiel. Un grand nombre d'expérimentations tend à montrer l'intérêt des idées proposées.

1 Introduction

Depuis quelques années maintenant, la résolution pratique du problème SAT est un domaine de recherche des plus actifs. Avec l'avènement des architectures multicœurs, ces recherches se concentrent plus particulièrement sur le développement de démonstrateurs parallèles, capables de traiter des problèmes applicatifs de grande taille. Parmi les solveurs parallèles disponibles, citons `ManySat` [10], `SArTagnan` [12], `Plingeling` [4], `ppfolio` [16], `part-tree-learn` [11].

*Supporté par le CNRS et OSEO, avec le projet ISI "Pajero".

Il existe principalement deux types d'approches pour résoudre SAT en parallèle : les approches de type « diviser pour régner » et les approches de type *portfolio*. Ces dernières, qui consistent à faire collaborer différentes recherches concurrentes, sont à l'heure actuelle les plus efficaces en pratique. Dans le cas particulier du problème SAT, la collaboration entre les différentes recherches est principalement réalisée par le biais d'échanges d'informations (souvent sous la forme de clauses apprises). Il est cependant extrêmement difficile de prédire quelles clauses sont les plus pertinentes à partager. Afin de palier ce problème, `ManySAT` limite l'échange de clauses en ne partageant que celles possédant une taille inférieure à une certaine limite [9]. Toutefois, malgré le fait que cette approche ait obtenu plusieurs distinctions lors des précédentes compétitions, la plupart des solveurs actuels (`Plingeling` [4], `SArTagnan` [12], etc.) se limitent au partage des clauses unitaires. En outre, lors de la dernière compétition, un nouveau solveur de type *portfolio*, nommé `ppfolio`, a obtenu de très bon résultats, glanant un grand nombre de récompenses : `ppfolio` est en fait un simple script qui exécute de manière indépendante différents solveurs séquentiels parmi les plus performants. Par conséquent, il semble que ne pas faire collaborer (ou très peu) soit un choix tout à fait acceptable d'un point de vue empirique. Ce constat suggère que les stratégies utilisées actuellement pour gérer l'échange d'informations entre les recherches concurrentes ne sont pas matures, et semblent pouvoir être améliorées.

Prédire l'utilité (future) d'une clause est également un problème connu pour la résolution de SAT dans le cadre séquentiel. En effet, les solveurs SAT modernes sont capables d'apprendre une clause après chaque conflit, et l'ajout de ces dernières a une conséquence directe sur l'efficacité du solveur, puisqu'elles influent sur les performances de la

propagation unitaire. Afin de palier ce problème, la base de clauses apprises est périodiquement purgée de certaines clauses jugées inutiles pour la suite de la recherche. Récemment, une nouvelle mesure nommée *psm* [1] a été proposée afin de gérer la base de clauses apprises. Cette mesure consiste à comparer l'interprétation (partielle) courante à l'ensemble des littéraux de chaque clause apprise. L'idée générale de cette approche est la suivante : si l'intersection entre l'interprétation courante et les littéraux d'une clause est grande, alors la clause sera probablement inutile dans le sous-espace de recherche dans lequel le solveur se trouve. Si au contraire le nombre de littéraux de l'intersection est petit, alors la clause a de fortes chances d'être utilisée dans le processus de propagation unitaire, et par conséquent de réduire l'espace de recherche. Cette nouvelle mesure a permis la mise en place d'une nouvelle stratégie de gestion de la base de clauses apprises. Cette dernière, contrairement aux autres politiques, permet de geler une clause lorsque celle-ci est considérée comme inutile dans l'espace de recherche courant. Ainsi, périodiquement les clauses apprises sont évaluées à l'aide de la mesure *psm*, afin d'en activer certaines et d'en geler d'autres. Différentes expérimentations ont permis de montrer que cette approche a un très bon comportement en pratique, et permet très souvent de sélectionner les clauses qui sont utiles dans un futur proche.

Dans ce papier, nous étendons les résultats présentés dans [1] à la résolution parallèle de SAT. Nous proposons différentes politiques d'exportation et d'importation de clauses, pour les différents *threads* d'un solveur *portfolio*. Cet article est structuré ainsi : la section suivante présente les notations et notions nécessaires à sa lecture, puis la section 3 présente quelques résultats préliminaires sur l'échange de clauses au sein d'un solveur parallèle de type *portfolio*. Ensuite, dans la section 4, nous présentons notre approche nommée `PeneLoPe`, puis étudions expérimentalement son comportement vis-à-vis des autres solveurs de l'état de l'art dans la section 5. Enfin, nous concluons en fournissant quelques perspectives.

2 Pré-Requis

Nous supposons le lecteur familier avec les notions liées au problème de satisfiabilité d'une formule propositionnelle (ie. variable x, littéral positif x ou négatif $\neg x$, clause, clause unitaire, interprétation et CNF). Nous rappelons juste brièvement le schéma général d'un solveur CDCL (*Conflict-Driven, Clause Learning*). Une branche d'un solveur CDCL est une séquence de décisions, suivies de propagations des littéraux unitaires, répétées jusqu'à ce qu'un conflit survienne. Chaque variable de décision choisie au moyen d'une heuristique, généralement basée sur l'activité, est affectée à un niveau de décision donné, les littéraux propagés en conséquence ayant le même niveau de décision. Chaque fois qu'un conflit survient, un *nogood* est calculé avec une méthode donnée, généralement celle nommée FUIP (*First Unique Implication Point*) [14, 19]. Le *nogood* est ajouté à la base des clauses apprises et un saut arrière est effectué. En outre, des redémarrages sont effectués périodiquement. Le lecteur désirant plus de détails sur les solveurs de type CDCL peut se référer à [7]. Dans la suite, certaines notions nécessaires à la compréhension du papier sont détaillées.

2.1 Gestion de la base de clauses apprises

La taille de la base de clauses apprises influe énormément sur les performances d'un solveur. En effet, conserver trop de clauses peut nuire à l'efficacité du processus de propagation unitaire (celui-ci étant intrinsèquement lié au nombre de clauses), tandis que supprimer trop de clauses peut faire perdre le bénéfice de l'apprentissage. Afin de palier ce problème, les solveurs effectuent périodiquement un nettoyage de la base qui consiste à supprimer les clauses apprises considérées comme inutiles. Par conséquent, identifier ce qu'est une bonne clause (ie. utile à l'établissement de la preuve) est clairement un challenge important. La première mesure de qualité proposée, certainement la plus populaire, est basée sur la notion d'activité inhérent à l'heuristique de choix de variable VSIDS. Plus précisément, une clause apprise est considérée comme utile à la preuve si elle apparaît souvent comme *raison* d'un conflit. Typiquement, cette stratégie de suppression considère qu'une clause utile dans le passé le sera dans le futur. Dans [2], une autre mesure, appelée *lbd*, est utilisée afin d'estimer la pertinence d'une clause. Cette nouvelle mesure est basée sur le nombre de niveaux de décision différents apparaissant dans une clause apprise lorsque que cette dernière est générée. Des expérimentations ont permis de mettre en exergue le fait que des clauses possédant une valeur de *lbd* faible sont plus souvent utilisées que celles avec une valeur de *lbd* élevée. Toutefois, bien que ces mesures soient efficaces en pratique, elles restent heuristiques et par conséquent ne peuvent éviter la suppression de clause utiles pour la suite de la recherche.

2.2 Solveur SAT parallèle

Il existe deux types d'approches pour résoudre SAT en parallèle. D'une part, les approches de type « diviser pour régner » consistent à diviser le problème en plusieurs sous-problèmes qui sont ensuite résolus indépendamment par différentes unités de calcul [5, 6]. L'un des inconvénients de ce modèle réside dans le partage de la charge de travail entre les processeurs. En effet, puisque la taille des sous-problèmes n'est pas connue à l'avance, il est nécessaire de mettre en place une politique de rééquilibrage entre les différentes unités de calcul. Ce rééquilibrage peut être fait de manière statique ou dynamique [11]. Notons que certaines

approches sont aussi capables de transférer des informations (le plus souvent sous la forme de *nogood*) entre les différentes unités de calcul [11, 17].

D'autre part, les approches de type *portfolio* mettent plusieurs solveurs en concurrence afin de résoudre le problème. Ces méthodes permettent ainsi d'exploiter la complémentarité entre différentes stratégies utilisées par les solveurs CDCL [10, 4, 12]. Puisque chaque processeur travaille avec la formule initiale, ces approches ne nécessitent pas la mise en place de stratégies d'équilibrage de charge, et la coopération est assurée par l'échange de clauses apprises par les différentes recherches. Malgré sa simplicité, ce type d'approche n'est pas toujours évident à mettre en place. En effet, le choix des stratégies utilisées est prépondérant (particulièrement si le nombre de processeurs disponibles est faible). De manière générale, l'objectif est de couvrir le plus de problèmes possibles.

Comme nous l'avons déjà souligné précédemment, la taille de la base de clauses apprises est un critère crucial pour l'efficacité des solveurs séquentiels. Par conséquent, dans le cadre d'une approche de type *portfolio*, il n'est pas souhaitable de partager systématiquement toutes les clauses apprises par les différentes recherches, au risque de voir la taille de la base de clauses apprises augmenter trop rapidement. Afin de conserver le coté collaboratif de ces approches, il est donc nécessaire de définir des critères afin d'identifier les clauses qui peuvent être échangées. Une manière naturelle de gérer cela consiste à considérer la taille des clauses comme critère de sélection. Les auteurs de `ManySAT` [10] proposent par exemple de ne partager que les clauses dont la taille est inférieure à 8. Cependant, après avoir observé que les clauses de petite taille apparaissent de moins en moins souvent durant la recherche, ils ont ensuite proposé de régler dynamiquement la taille limite des clauses échangées entre les différents *threads* [9]. Toutefois, il est surprenant qu'un solveur tel que `Plingeling` qui n'échange entre les *threads* que les clauses unitaires soit l'un des vainqueurs de la compétition SAT 2011. Plus surprenant encore : le solveur `ppfolio`, qui a obtenu un nombre important de récompenses lors de cette même compétition, n'échange aucune information et ne fait qu'exécuter en parallèle -et de manière indépendante- plusieurs solveurs séquentiels. Ces dernières observations montrent à quel point les techniques liées à la collaboration entre les recherches semblent pouvoir être améliorées.

3 Expérimentations préliminaires

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

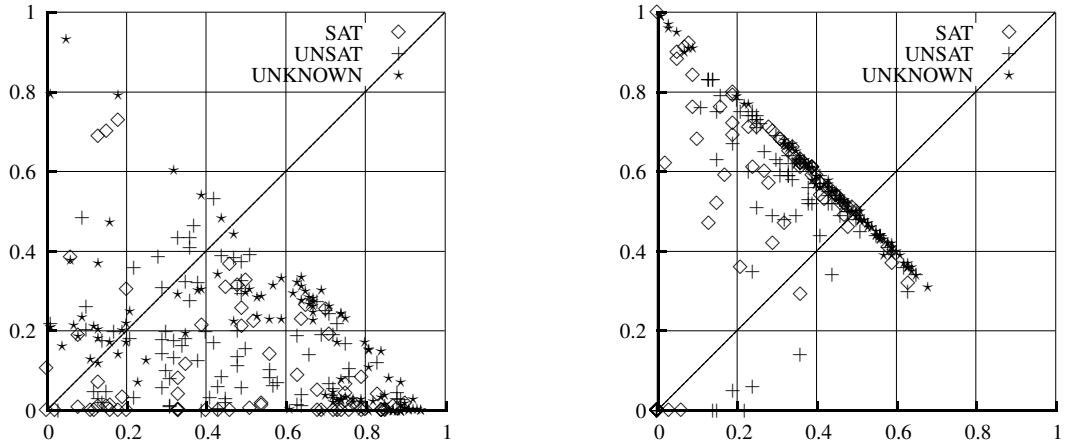
Pour cela, nous avons mené des expérimentations [1] sur `ManySAT 2.0` (basé sur `Minisat 2.2`), l'un des solveurs *portfolio* les plus efficaces, et au sein duquel une vraie collaboration est réalisée. Ce solveur ne différencie les *threads* que sur les premières variables de décision, qui sont choisies au hasard. Ainsi, exceptée l'interprétation initiale, chaque *thread* associé à un solveur CDCL possède exactement le même comportement (en terme de redémarrages, heuristique de choix de variable, etc.), ce qui permet de faire une comparaison plus juste, limitant les effets de bord liés aux autres paramètres. Par défaut, toutes les clauses apprises de taille inférieure à 8 sont partagées entre les *threads*. De plus, `ManySAT` possède un mode déterministe [8] que nous avons utilisé afin de rendre reproductibles les différents résultats proposés dans cet article [2].

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

La Figure 1 illustre les résultats obtenus par les différentes instances. Ils y sont reportés de la manière suivante : chaque point de la figure correspond à une instance, l'axe des x représentant le taux d'utilisation des clauses partagées ($\#used(\mathcal{SC})/\#\mathcal{SC}$) tandis que l'axe des y représente celui des clauses inutiles et supprimées ($\#unused(\mathcal{SC})/\#\mathcal{SC}$) et nous reportons la moyenne sur les différents *threads*. La Figure 1(a) donne les résultats pour `ManySAT`. Nous pouvons tout d'abord remarquer que le taux d'utilisation des clauses partagées diffère fortement d'une instance à l'autre. Nous pouvons également remarquer que dans de nombreux cas, `ManySAT` conserve

1. Toutes les expérimentations conduites dans cet article sont réalisées sur des bi-processeurs `Intel XEON X5550` 4 cœurs à 2.66 GHz avec 8Mo de cache et 32Go de RAM, sous Linux CentOS 6. (kernel 2.6.32). Chaque solveur utilise 8 *threads*. Le temps limite alloué pour résoudre une instance est de 1200 secondes WC. Nous avons utilisé les 300 instances de la catégorie *application* de la compétition SAT 2011.

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



(a) `ManySAT` (b) `ManySAT` aggressif

FIGURE 1 – Comparaison entre les clauses partagées utilisées et les clauses partagées non utilisées et supprimées. Chaque point correspond à une instance. L'axe des x donne le taux d'utilisation des clauses partagées $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, alors que l'axe des y donne celui des clauses partagées non utilisées et supprimées $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

des clauses pendant toute la recherche (points proches de l'axe des x). Ceci est dû à la politique non agressive de `Minisat` où dans de nombreux cas, très peu de nettoyages de clauses sont effectués. Les différents *threads* peuvent donc conserver les clauses inutiles durant une longue période, causant un sur-coût sans engendrer de bénéfice.

Nous avons réalisé les mêmes expérimentations en utilisant une politique de nettoyage des clauses beaucoup plus agressive, a savoir celle utilisée dans [1] (voir la section 4) et nous reportons les résultats dans la Figure 1(b). Ici, pour de nombreuses instances, les clauses sont supprimées avant d'avoir été utilisées et le taux de clauses utiles décroit fortement par rapport à la version basique de `ManySAT`.

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

Nous pouvons également noter que l'utilisation du *lbd* pour mesurer la qualité des clauses apprises ne semble pas non plus adéquate. En effet, les clauses partagées sont de petites clauses et ont donc une petite valeur de *lbd*. Les clauses importées ont alors plus de chance d'être préférées aux clauses générées par le *thread* lui-même. Ainsi, même s'il semble possible de paramétrer plus précisément la stratégie de nettoyage afin d'obtenir un solveur plus robuste, nous pensons que la gestion actuelle des clauses apprises n'est pas appropriée dans le cas du partage de clauses des solveurs *portfolio*. Nous proposons donc une nouvelle stratégie dans la prochaine section.

4 Sélectionner, partager et activer les bonnes clauses

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

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

Pour chacune de ces raisons, nous proposons d'utiliser la politique de gestion dynamique des clauses apprises proposée par [1] au sein de chaque fil d'exécution. Cette technique récente permet d'activer ou de geler certaines clauses apprises, importées ou générées localement. L'avantage de cette technique est double. Étant donné qu'elles peuvent

être gelées, la surcharge calculatoire occasionnée par les clauses importées est grandement réduite. De plus, les clauses importées, qui peuvent se révéler utiles dans un futur plus ou moins proche de la recherche, sont activées au moment adéquat. La prochaine section présente de manière plus précise cette méthode.

Geler certaines clauses

La stratégie proposée dans [1] est très différente de celles proposées par le passé (voir section 2). Elle est en effet basée sur le gel et l'activation dynamique des clauses apprises. À un certain point de la recherche, les clauses les plus prometteuses sont activées tandis que les autres sont gelées. De cette façon, les clauses apprises peuvent être écartées pour les prochaines propagations, mais peuvent par la suite être réactivées. Cette stratégie ne peut être utilisée avec les mesures d'activité basées sur le VSIDS ou le *lbd*. En effet, la mesure d'activité (inspirée par VSIDS) est dynamique mais ne peut être utilisée que pour mettre à jour l'activité des clauses actives -c'est-à-dire participant effectivement à la recherche- , tandis que *lbd* est soit statique (et ne change donc pas durant la recherche), soit dynamique auquel cas on rencontre le même problème que la mesure basée sur VSIDS. De ce fait, cette stratégie est associée avec une autre mesure afin de mesurer l'utilité d'une clause apprise [1].

Soient c une clause apprise par le solveur et ω l'interprétation courante résultant de la dernière polarité des variables de décisions [15]. La valeur *psm* de la clause c par rapport à ω, notée $psm_\omega(c)$, est égale à $psm_\omega(c) = |\omega \cap c|$.

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

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

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

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

Nous présentons donc une technique où l'expéditeur *et* le destinataire peuvent avoir leur propres stratégies. Chaque expéditeur (stratégie d'export) essaie de trouver dans sa base de clauses apprises les informations les plus pertinentes pour aider les autres *threads*. De plus, le destinataire (stratégie d'import) peut ne pas accepter aveuglément les clauses qui lui sont envoyées. Nous avons nommé ce solveur prototype `PeneLoPe`[3] (**P**arallel **L**bd **P**sm solver).

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

- *no-freeze* : chaque clause importée est directement activée, et sera évaluée (et potentiellement gelée) durant le prochain appel à *updateDB* .
- *freeze-all* : chaque clause importée est *gelée* par défaut, et ne sera utilisée par la suite que si elle remplit les conditions pour être activée.
- *freeze* : chaque clause est évaluée au moment de l'import. Elle est activée si elle est considérée comme prometteuse, selon les mêmes critères que si elle avait été générée localement.

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

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

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

TABLE 1 – Comparaison entre les différentes stratégies d'import, d'export & redémarrage, en utilisant le mode déterministe

- *unlimited* : chaque clause générée est exportée aux différents fils d'exécution.
- *size limit* : seules les clauses dont la taille est inférieure à une valeur donnée (8 dans nos expérimentations) sont exportées [10].
- *lbd limit* : une clause est exportée aux autres fils d'exécution si son *lbd* est inférieur à une limite donnée d (8 par défaut). Il est important de remarquer que la valeur du *lbd* peut être réduite au cours du temps. Ainsi, dès que $lbd(c)$ est inférieur à d, la clause est exportée.

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

- *Luby* : Soit l_n le n-ième terme de la série Luby [13]. Le n-ième redémarrage est effectué après $l_n \times \alpha$ conflits (α vaut 100 par défaut).
- *lbd* : Soit LBD_g la moyenne des valeurs de *lbd* de chaque clause apprise depuis le commencement de la recherche. Soit LBD_{100} la moyenne des 100 dernières clauses apprises. Cette politique induit qu'un redémarrage est effectué dès que $LBD_{100} \times \alpha > LBD_g$ (α vaut 0.7 par défaut) [2].

Différentes expérimentations ont été réalisées dans le but de comparer ces politiques d'import, d'export et de redémarrage. Différentes versions ont été exécutées et la table 1 présente une partie des résultats obtenus. Cette table rapporte pour chaque stratégie le nombre d'instances SAT (#SAT), UNSAT (#UNSAT) et total (#SAT + #UNSAT) résolues.

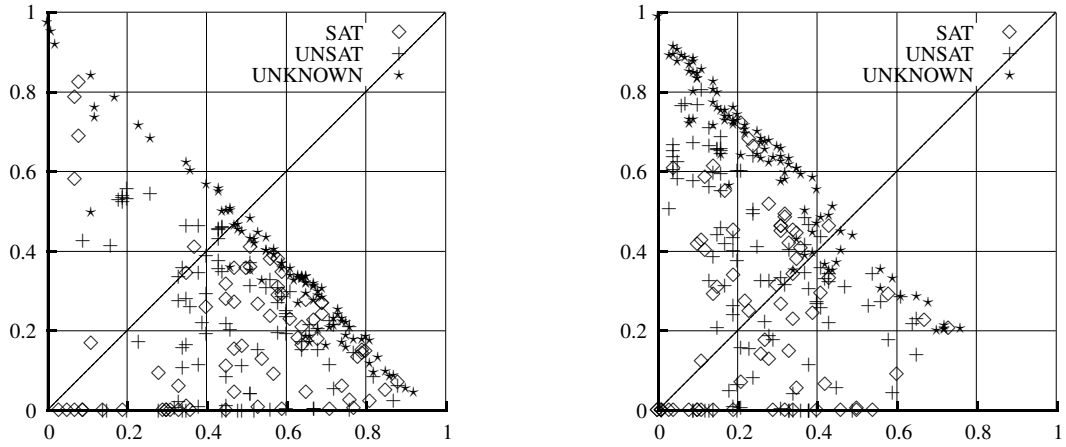
Comparons d'abord la stratégie d'import. Sans surprise, la politique *unlimited* obtient les plus mauvais résultats. En effet, aucune version utilisant cette stratégie n'est capable de résoudre plus de 190 instances. Chaque clause générée est exportée, et nous obtenons ainsi le niveau maximum de communication.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

C'est la raison pour laquelle des limites basées sur la taille ont été introduites avec l'idée que de petites clauses permettent de mieux filtrer l'espace de recherche et sont donc préférables. En effet, on constate dans la table 1 que la politique *size limit* surpasse *unlimited*. De plus, il est également possible de constater que l'échange des seules clauses unitaires n'offre pas d'aussi bonnes performances (*size = 1*), tout comme cela avait été annoncé préalablement. Il est également important de signaler que les clauses de grande taille peuvent grandement réduire la taille de la preuve [3].

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

Ceci pourrait représenter un problème pour un solveur parallèle n'ayant pas la possibilité de geler certaines clauses. Cependant, puisque `PeneLoPe` contient un tel mécanisme, l'impact est grandement réduit. D'un point de vue empirique, la table 1 montre que *lbd limit* obtient les meilleurs résultats entre les différentes politiques.

Concentrons nous maintenant sur la politique de redémarrage. De façon évidente, la politique *luby* obtient globalement de moins bons résultats que celle utilisant *lbd*. Cela montre clairement l'intérêt de cette mesure introduite dans [2]. À propos de la stratégie d'import, aucune stratégie ne se révèle être clairement meilleure que les autres. En effet, le meilleur résultat relatif au nombre d'instances SAT est obtenu par *no freeze* (97) lorsqu'il est associée à la politique de redémarrage *luby* et *size limit* comme politique d'export, tandis que le meilleur résultat en nombre d'instance UNSAT est obtenu en utilisant la stratégie *freeze* (113). De plus, *no freeze* permet d'obtenir les meilleurs résultats globaux en résolvant 205 instances parmi les 300



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

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

FIGURE 2 – Comparaison entre les clauses importées révélées utiles et les clauses importées n'ayant pas été utilisées dans la propagation et supprimées. Chaque point représente une instance. L'axe x représente le taux de clauses utiles $\frac{\#used(\mathcal{SC})}{\#\mathcal{SC}}$, tandis que l'axe y représente le taux de clauses non utilisées et supprimées $\frac{\#unused(\mathcal{SC})}{\#\mathcal{SC}}$.

utilisées. Il serait ainsi audacieux de plaider envers l'une des 3 techniques proposées. Cependant, un grand nombre des politiques proposées obtiennent de meilleurs résultats que les techniques "classiques" d'échange de clauses, représentées dans la table 1 par `ManySat`.

Dans une seconde expérimentation, nous avons voulu évaluer le comportement de `PeneLoPe` avec certaines des politiques proposées. Dans ce but, nous avons reconduit le même type d'expérimentations que celle présentée dans la section 3 ; les résultats obtenus sont reportés dans la Figure 2. Dans un premier temps, nous avons essayé avec les politiques *size limit*, *luby*, et *no freeze* (dénoté `SLN`, voir Figure 2(a)).

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

Ensuite, l'expérimentation fut reconduite en utilisant les politiques *lbd limit*, *lbd* , et *freeze* (dénoté `LLF`, voir Figure 2(b)). Au premier abord, le comportement de cette version est moins satisfaisant que la version `SLN`, du fait que pour la plupart des instances, plus de la moitié des clauses importées sont supprimées sans avoir été utiles dans la propagation. En réalité, dans cette version, un nombre bien plus important de clauses sont exportées du fait de la politique d'export *lbd limit*, ce qui conduit à un taux d'utilisation plus faible. Un réglage plus fin des paramètres (limite d'export pour les valeurs *lbd*, nombre de fois qu'une clause peut être gelée avant d'être supprimée, etc.) pourrait améliorer cela.

Si l'on regarde les statistiques détaillées de quelques instances, présentées dans la table 2, il apparaît que la version `LLF` partage en effet beaucoup plus de clauses que la version `SLN` (colonne nb_u). Notons aussi que cette table contient d'autres informations très intéressantes. Par exemple, il est possible de voir que pour certaines instances (par exemple `AProVE07-21`), quasiment 90% des clauses importées sont en fait gelées et ne participent pas immédiatement à la recherche, alors que pour d'autres instances la situation inverse se produit (`hwmcc...`). Ceci révèle la grande adaptabilité de la mesure *psm*.

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

5 Comparaison avec les solveurs états de l'art

Dans cette section, nous proposons une comparaison de deux de nos prototypes avec les meilleurs solveurs SAT parallèles connus à ce jour. Nous avons ainsi sé-

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

TABLE 2 – Statistiques à propos d'instances UNSAT. Pour chacune d'entre elle et pour chaque version de PeneLoPe, nous reportons le temps WC nécessaire pour résoudre l'instance, le nombre de conflits (nb_c, en milliers), le nombre de clauses importées (nb_i, en milliers) avec entre parenthèses le taux entre nb_i et nb_c, le pourcentage de clauses gelée à l'import (nb_f), le pourcentage de clauses importées utiles (nb_u) et le pourcentage de clauses importées et supprimées sans avoir été utiles dans la recherche (nb_d). À l'exception du temps, chaque mesure est une moyenne sur les 8 fils d'exécution.

lectionné les solveurs qui se sont montrés les plus efficaces lors des dernières compétitions : ppfolio [16], cryptominisat [18], plingeling [4] et ManySat [10]. Pour PeneLoPe, nous avons choisi pour les deux versions la stratégie de redémarrage basée sur *lbd*, ainsi que la politique d'exportation *lbd limit*. Ces versions ne diffèrent donc que par leur politique d'importation de clauses, dont l'une est *freeze*, l'autre *no freeze*. Précisons également que contrairement à tous les résultats présentés précédemment, nous n'avons pas utilisé le mode déterministe dans cette partie, dans le but d'obtenir les meilleures performances possibles.

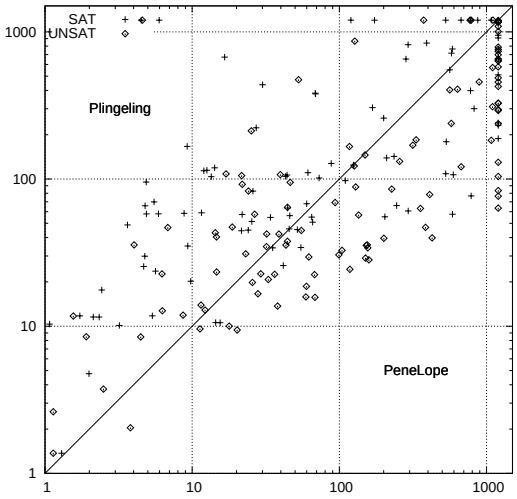
La Figure 3 donne les résultats selon différentes représentations. PeneLoPe dépasse en pratique tous les autres solveurs paralèlles. En effet, il parvient à résoudre 216 instances tandis qu'aucun autre solveur ne dépasse les 200 (Figure 3(a)). Notons cependant que si on ne considère que les instances satisfaisables, les meilleurs résultats sont obtenus par plingeling qui en résout 99. Ceci est particulièrement visible dans la Figure 3(b) où PeneLoPe et plingeling sont plus précisément comparés. Dans cette Figure, la plupart des points représentant des instances SAT sont assurément au dessus de la diagonale, illustrant la force de plingeling sur ce type de problèmes. Toutefois, les résultats relatifs aux instances SAT sont assez proches les uns des autres (97 pour PeneLoPe *freeze*, 95 pour ManySat, etc.), l'écart étant plus important pour les problèmes UNSAT.

En outre, nous avons également comparé ces solveurs sur une architecture composée de 32 cœurs. Plus précisément, la configuration matérielle est maintenant la suivante : Intel Xeon CPU X7550 (4 processeurs, 32 cœurs) 2.00GHz avec 18 Mo de mémoire cache et 256Go de mémoire vive. Chaque solveur est exécuté avec 32 *threads*, et les résultats obtenus sont présentés dans la Figure 4 de manière similaire.

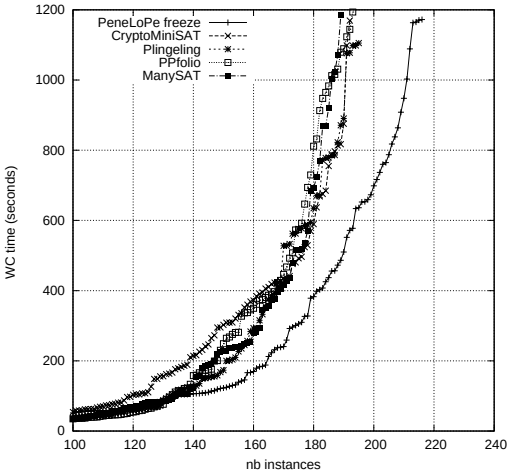
Tout d'abord, notons qu'à l'exception de plingeling, tous les solveurs améliorent leurs performances quand ils sont exécutés avec un nombre supérieur de *threads*. Le profit est cependant limité pour certains d'entre eux. Par exemple, cryptominisat résout 193 instances avec 8 *threads*, et 201 instances avec 32 *threads*. L'amélioration est bien supérieure avec PeneLoPe, dont les deux versions résolvent 15 instances supplémentaires, et plus spectaculaire encore pour ManySAT avec un gain de 29 instances. L'écart est tout particulièrement visible sur la Figure 4(c), puisque nos 3 compétiteurs résolvent le même nombre d'instances sur (environ) le même temps (les courbes de ppfolio, Plingeling et cryptominisat sont très proches les unes des autres), tandis que la courbe de PeneLoPe et celle de ManySAT montrent clairement leur capacité à résoudre un plus grand nombre de problèmes en un temps restreint. Il est d'ailleurs bon de noter que PeneLoPe résout le même nombre de problèmes que Plingeling, ppfolio et cryptominisat avec une limite (virtuelle) de temps de seulement 400 secondes. Enfin, il semble que le comportement de PeneLoPe puisse être amélioré sur les instances SAT. En effet, il apparaît que les redémarrages *luby* sont plus efficaces pour les problèmes possédant une solution que pour ceux qui en sont dépourvus, alors que le contraire se produit pour le redémarrage *lbd*.

Solver	#SAT	#UNSAT	#SAT+#UNSAT
PeneLoPe *freeze*	97	**119**	**216**
PeneLoPe *no freeze*	96	**119**	215
plingeling [4]	**99**	97	196
ppfolio [16]	91	103	194
cryptominisat [18]	89	104	193
ManySat [10]	95	92	187

(a) PeneLoPe VS l'état de l'art



(b) PeneLoPe *freeze* VS Plingeling



(c) Cactus plot

FIGURE 3 – Comparaison sur 8 cœurs

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

Pour finir, insistons sur le fait que durant nos expérimentation avec PeneLoPe, tous les *threads* possèdent *exactement* les mêmes paramètres et les mêmes stratégies, comme dans nos expérimentations préliminaires présentées dans la section 3. Offrir de la diversification aux différentes recherches CDCL concurrentes devrait accroître plus encore l'efficacité pratique de notre prototype.

6 Conclusion

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

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

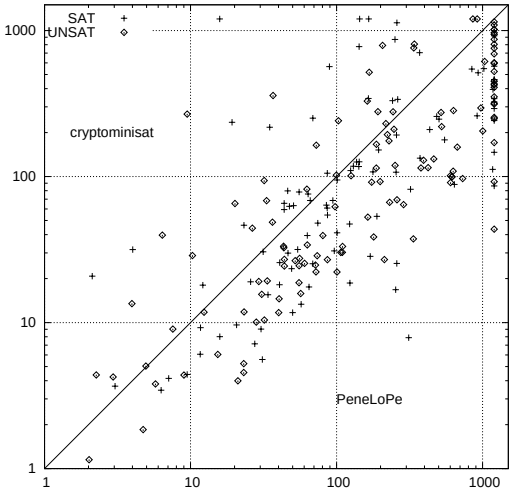
Références

[1] G. Audemard, J.M. Lagniez, B. Mazure, and L. Saïs. On freezeing and reactivating learnt clauses. Dans *proceedings of SAT*, pages 147–160, 2011.

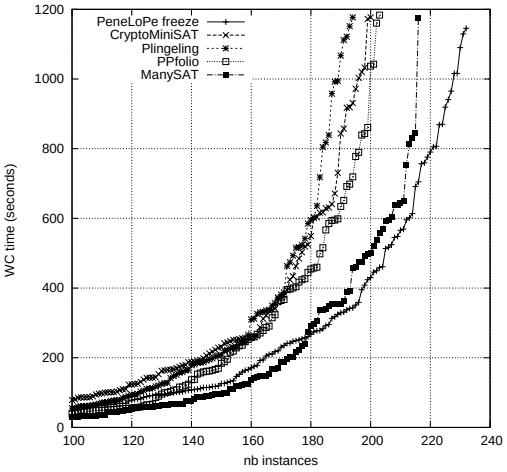
[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. Dans *proceedings of IJCAI*, pages 399–404, 2009.

Solver	#SAT	#UNSAT	#SAT+#UNSAT
PeneLoPe *freeze*	104	127	**231**
PeneLoPe *no freeze*	99	**131**	230
ManySAT [10]	105	111	216
ppfolio [16]	**107**	97	204
cryptominisat [18]	96	105	201
Plingeling [4]	100	95	195

(a) PeneLoPe VS l'état de l'art



(b) PeneLoPe *freeze* VS cryptominisat



(c) Cactus plot

FIGURE 4 – Comparaison sur 32 cœurs

[3] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. Dans *JAIR*, 22 :319–351, 2004.

[4] A. Biere. (p)lingeling. http://fmv.jku.at/lingeling.

[5] W. Chrabakh and R. Wolski. GrADSAT : A parallel SAT solver for the grid. Rapport technique, 2003.

[6] G. Chu, P. Stuckey, and A. Harwood. Pminisat : a parallelization of minisat 2.0. Rapport technique, SAT Race, 2008.

[7] A. Darwiche and K. Pipatsrisawat. *Complete Algorithms*, chapter 3, pages 99–130. IOS Press, 2009.

[8] Y. Hamadi, S. Jabbour, C. Piette, and L. Saïs. Deterministic parallel DPLL. Dans *JSAT*, 7(4) :127–132, 2011.

[9] Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. Dans *proceedings of IJCAI*, pages 499–504, 2009.

[10] Y. Hamadi, S. Jabbour, and L. Saïs. ManySat : a parallel SAT solver. Dans *JSAT*, 6 :245–262, 2009.

[11] A. Hyvärinen, T. Junttila, and I. Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. Dans *proceedings of CP*, pages 385–399, 2011.

[12] S. Kottler and M. Kaufmann. SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. Dans *Pragmatics of SAT*, 2011.

[13] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. Dans *proceedings of ISTCS*, pages 128–133, 1993.

[14] J. Marques-Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. Dans *proceedings of ICCAD*, pages 220–227, 1996.

[15] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. Dans *proceedings of SAT*, pages 294–299, 2007.

[16] O. Roussel. ppfolio. http://www.cril.univ-artois.fr/~roussel/ppfolio.

[17] T. Schubert, M. Lewis, and B. Becker. Pamiraxt : Parallel sat solving with threads and message passing. Dans *JSAT*, 6(4) :203–222, 2009.

[18] Mate Soos. Cryptominisat. http://www.msoos.org/cryptominisat2/.

[19] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. Dans *proceedings of ICCAD*, pages 279–285, 2001.