Concilier Parallélisme et Déterminisme dans la Résolution de SAT

Youssef Hamadi [1,2] **Said Jabbour** [3] **Cédric Piette**[3] **Lakhdar Saïs**[3]

[1] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom

[2]LIX École Polytechnique, F91128 Palaiseau, France

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

youssefh@microsoft.com {jabbour,piette,sais}@cril.fr

Résumé

La résolution parallèle du problème SAT est maintenant un important champ de recherche. Le vaste déploiement de plateformes multi-cœurs combinés à la disponibilité d'instances SAT difficiles sont probablement à l'origine de ce succès. Toutefois, les solveurs SAT parallèles proposés jusqu'à présent souffrent d'un comportement non déterministe. Ceci est la conséquence de leur architecture qui n'effectue que peu (ou pas) de synchronisation, dans le but d'optimiser les performances. Ce caractère non déterministe est un revers pour les utilisateurs, et nombre d'applications nécessitent la reproductivité des résultats, à la fois en terme de temps d'exécution et de solutions retournées. Dans cet article, nous proposons le premier solveur SAT parallèle et déterministe (modulo le nombre de cœurs disponibles), basé sur un portfolio de solveurs SAT synchronisés. Nos résultats expérimentaux montrent clairement que notre approche préserve les performances d'un solveur non déterministe tout en garantissant une reproductibilité complète des résultats.

1 Introduction

La résolution parallèle du problème SAT a reçu énormément d'attention ces 3 dernières années. Ce phénomène est dû à plusieurs facteurs, comme la large disponibilité d'architectures multi-cœurs, ou la relative stagnation des performances des solveurs SAT séquentiels. Malheureusement, la supériorité des solveurs parallèles vient au prix de résultats non reproductibles, à la fois en terme de temps d'exécution et de solutions retournées. Ce comportement est la conséquence de leur architecture qui n'effectue aucune synchronisation dans l'objectif de maximiser les performances. Aujourd'hui, SAT est la pierre angulaire de nombreuses applications industrielles, telle que la vérification formelle, la démonstration automatique ou la bioinformatique. Ces domaines nécessitent la reproductibilité des résultats, et ne peuvent donc tirer parti de l'efficacité des solveurs parallèles. A titre d'exemple, un outil de vérification logiciel basé sur un solveur SAT parallèle peut retourner différents bugs s'il est exécuté plusieurs fois sur un même morceau de code source ; une situation inacceptable pour un ingénieur spécialisé en vérification.

Considérons un autre exemple : un utilisateur d'une application de conception d'emploi du temps basé un solveur SAT parallèle peut être étonné de voir générer des emplois du temps différents, avec pourtant le même ensemble de contraintes, via des appels successifs du solveur. Avec de telles applications, la reproductivité est préférable, sinon nécessaire. En outre, l'intégration de composants non déterministes tels qu'un solveur parallèle dans un logiciel augmente inévitablement le coût de la phase de débuggage. Ceci limite clairement le déploiement des technologies de parallélisation pour SAT.

Plus largement, en science comme en ingénierie, la re-

productibilité des résultats est fondamentale, et dans le but de servir à un nombre toujours plus grand d'applications, les solveurs parallèles doivent donc être déterministes. En effet, évaluer ou comparer des solveurs SAT non déterministes est clairement problématique. Ce problème a conduit les organisateurs des compétitions SAT et SAT-Race à mentionner clairement leur préférence pour les solveurs déterministes et d'introduire des règles spécifiques pour l'évaluation des solveurs SAT parallèles. On peut par exemple trouver ceci dans le règlement de la SAT-Race 2008 : « dans le but d'obtenir des résultats reproductibles, les solveurs SAT doivent s'abstenir autant que possible d'utiliser des constructions non déteministes. Pour les implantations parallèles, il est pourtant difficile d'obtenir des temps d'exécutions reproductibles. Ainsi, cette condition ne s'applique pas à la catégorie parallèle. Comme les temps d'exécution des solveurs paralèlles peuvent très sensiblement varier, chaque solveur est exécuté trois fois sur chaque instance. Une instance est considérée résolue si au moins l'une des trois exécutions du solveur a permis de conclure quant à sa satisfaisabilité ». Pour les catégories parallèles, des règles d'évaluation différentes ont été utilisées lors de la compétition SAT 2009, ainsi que lors de la SAT Race 2010, illustrant les problèmes inhérents au non déterminisme dans l'évaluation des solveurs SAT parallèles.

Dans ce papier, nous proposons un solveur SAT à la fois parallèle et déterministe. Ces résultats sont intégralement reproductibles, à la fois en terme de temps d'exécution et de solutions retournées, et son efficicacité est équivalente à un solveur parallèle non déterministe. Celui-ci est basé sur un contrôle des intéractions entre les différents fils d'exécution à travers l'utilisation de barrières de synchronisation. Pour maximiser son efficacité, l'échange d'informations (clauses conflits) et la vérification de terminaison sont effectués sur une base régulière. Comme nous le montrons plus loin, la fréquence de ces échange influence largement les performances de notre solveur parallèle. Ce papier explore le compromis nécessaire entre de fréquentes synchronisations, qui permettent une intégration rapide des clauses apprises par les autres fils d'exécution au prix d'une attente importante, et la synchronisation plus sporadique, qui évite le coût élevé de l'attente entre fils d'exécutions mais retarde l'importation de clauses conflits.

Ce papier est organisé comme suit : dans la section suivante, nous détaillons les caractéristiques principales des solveurs SAT séquentiels et parallèles, et fournissons la motivation de ce travail dans la section 3. Notre contribution principale est détaillée dans la section 4, et est expérimentalement évaluée dans la section 5. Enfin, nous concluons par quelques pistes intéressantes liées à ce travail.

2 Préliminaires

2.1 Solveurs SAT modernes

Le problème SAT consiste à trouver une affectation à chaque variable d'une formule propositionnelle exprimée sous forme normale conjonctive (CNF). La plupart des solveurs modernes sont basés sur une version avancée de la procédure de Davis, Putnam, Logemann and Loveland, appelée communément DPLL [9]. Un solveur SAT à la DPLL effectue une recherche arborescente, selectionnant à chaque nœud de l'arbre un littéral *de décision* qu'il affecte à une valeur booléenne. Cette affectation est suivie d'une étape d'inférence qui déduit et propage certains littéraux, dits *unitaires*. Ces opérations sont enregistrées dans un *graphe d'implication*, une structure de données centrale qui stocke l'affectation partielle courante, ainsi que ses implications unitaires. Ce *processus de branchement* est répété jusqu'à ce que soit un modèle soit trouvé, soit un conflit soit atteint. Dans le premier cas, la formule est prouvée satisfaisable, et le modèle généré est retourné, alors que dans l'autre cas, une *clause conflit* (également appelée *clause assertive*) est générée par résolution, suivant un parcours du graphe d'implication [16, 20]. Ce *processus d'apprentissage* s'arrête dès qu'une clause ne contenant qu'un seul littéral non affecté (par rapport au niveau de décision courant) est obtenue. Une telle clause conflit exprime le fait que ce littéral est impliqué à un niveau supérieur de l'arbre. Le solveur effectue donc un saut en arrière (backtrack) à ce niveau d'implication et affecte ce littéral de manière à le satisfaire. Si une clause conflit vide est générée, alors la formule est prouvée insatisfaisable.

Au-delà de ce schéma de base, les solveurs s'appuient sur des techniques additionnelles, telles que les heuristiques basées sur l'activité, ou les potiliques de redémarrage. Détaillons un peu ces deux caractéristiques importantes. L'activité de chaque variable rencontrée lors du processus d'apprentissage est incrémentée. La variable ayant la plus importante activité est sélectionnée comme prochaine variable de décision. Ceci correspond à l'heuristique de choix de variable VSIDS (Variable State Independent Decaying Sum) [20]. En outre, après qu'un certain nombre de conflits soit atteint, la recherche est redémarrée. L'évolution de cette limite de conflits est contrôlée par une politique de redémarrage (voir par exemple [1]). Grâce à l'activité des variables enregistrées durant les précédentes exécutions partielles, l'objectif du redémarrage est de se concentrer tôt dans la recherche (i.e. en haut de l'arbre de recherche) sur des parties importantes de la formule. Ceci diffère des objectifs initiaux de la randomisation et des redémarrages introduits dans [11]. Pour un panorama plus exhaustif des techniques actuelles pour la résolution de SAT, nous conseillons la lecture de [4].

2.2 Solveurs SAT parallèles

Il existe principalement deux approches de résolution parallèle en programmation par contraintes. La première implante l'idée de « diviser pour régner », qui fractionne incrémentalement l'espace de recherche en sous-espaces, successivement alloués à différents travailleurs séquentiels [8, 17, 7, 10, 15, 5]. Ces travailleurs coopérent à travers une stratégie qui rééquilibre et réalloue dynamiquement les sous-espaces aux travailleurs à l'arrêt, ainsi que par l'échange de nogood [18, 6, 7].

L'approche à base de portfolio a été introduite en 2008 pour le problème SAT. Elle exploite la complémentarité entre plusieurs recherches DPLL séquentielles concurrentes et les fait coopérer. Puisque dans ce cas chaque recherche parallèle est effectuée sur la même formule, il n'y a pas besoin d'implanter des stratégies de réallocation dynamique du travail, car la coopération n'est effectuée qu'à travers l'échange de clauses. Avec ce type d'approches, un paramétrage fin des stratégies de recherche est essentiel, plus encore avec un faible nombre d'unités de calcul. De manière générale, l'objectif est de couvrir l'espace des *bonnes* stratégies de recherche de la meilleure façon possible.

Durant ces deux dernières années, les solveurs SAT parallèles basés sur des portfolios sont devenus proéminents ; à notre connaissance, durant cette période, aucune approche de type « diviser pour règner » n'a été développée (la dernière en date étant [7]). Nous décrivons ici les différents solveurs parallèles qualifiés lors de la SAT Race 2010 [1]. Ces implantations représentent d'état de l'art en terme de solveurs SAT parallèles.

Le vainqueur de la catégorie parallèle de la SAT Race 2010, appelé `plingeling` [3], dupplique l'instance à résoudre via un fil maître et l'alloue à des fils travailleurs. Les stratégies des travailleurs se différencient principalement par la quantité de prétraitement, la graîne aléatoire et l'heuristique de choix de variables. Le partage de clauses est réduit aux clauses unitaires qui sont échangées à travers le fil maître.

`ManySAT` [12] est historiquement le premier solveur SAT portfolio. Il duplique l'instance à résoudre, et exécute différents solveurs indépendants qui se différencient par leur politique de redémarrage, leur heuristique de branchement, leur graîne aléatoire et leur technique d'apprentissage. L'échange de clauses est également effectué suivant différentes politiques. Deux versions de ce solveur ont été soumises à la SAT Race 2010, et ont terminé deuxième et troisième.

De manière similaire à `ManySAT`, le solveur `SArTagnan` [14] exploitent différentes recherches arborescentes principalement différenciées par leur politique de redémarrage, et leur heuristique inspirée de VSIDS. Certains fils appliquent un processus de résolution dynamique [2], ou exploitent la notion de « points de référence » [13]. D'autres fils essaient de simplifier la base de clauses apprises en effectant des techniques d'élimination de variables. Ce solveur a fini quatrième.

Enfin, le solveur `Antom` [19] différencie ses fils d'exécution par l'heuristique de choix de variable, la stratégie de redémarrage, la détection de clause conflit et l'hyper résolution binaire effectuée de manière paresseuse [2], ainsi que la propagation unitaire dynamique. Le partage de clauses conflits est implémentée. Ce solveur a fini cinquième.

3 Motivations

Comme mentionné plus haut, tous les solveurs SAT parallèles souffrent d'un comportement non déterministe. Ceci est la conséquence de leur architecture qui n'effectue que peu (ou pas) de synchronisation, dans le but de maximiser les performances. Le partage de clauses conflits, généralement utilisé pour améliorer leurs performances, accentue plus encore ce comportement non déterministe, c'est-à-dire la non-reproductibité en terme de temps d'exécution, de solution retournée ou de preuve d'insatisfaisabilité.

Pour illustrer les variations en terme de solutions et de temps d'exécution, nous avons exécuté à de multiples reprises le même solveur parallèle (`ManySAT` version 1.1 [12]) sur toutes les instances satisfaisables utilisées pour la SAT Race 2010, et nous reportons certaines mesures. Chaque instance a été résolue 10 fois, et le temps limite pour chacune des exécutions a été fixé à 40 minutes. Les résultats obtenus sont reportés dans la Table 2.2. Pour chaque instance, nous donnons le nombre de variables ($\#vars$), le nombre d'exécutions réussies ($\#models$) et le nombre de modèles différents (*diff*) obtenus par ces exécutions. Deux modèles sont différents si leur distance de Hamming n'est pas égale à 0. Nous reportons également la distance moyenne de Hamming normalisée $n\bar{H} = \frac{\bar{H}}{\#vars}$ où $\bar{H}$ est la distance de Hamming moyenne entre chaque paire de modèles obtenus.

Comme on pouvait s'y attendre, le solveur fait preuve d'un comportement non déterministe en terme de modèles retournés. En effet, sur de nombreuses instances, il est possible d'observer que chaque exécution conduit à un modèle différent. Ces différences sont illustrées dans de nombreux cas par une importante valeur pour $n\bar{H}$. Par exemple, pour la famille $sha0_*_*$, le nombre de modèles différents est de 10, et $n\bar{H}$ vaut environ 30%, signifiant que les modèles diffèrent de 30% de leur valeur de vérité en moyenne. Enfin, le temps moyen (tpsMoy) et la déviation standard (σ) montrent la variabilité du solveur parallèle en terme de temps d'exécution. L'instance $12pipe_bug8$ illustre un cas extrême de ce phénomène ; en effet, pour l'obtention d'un

1. `http://baldur.iti.uka.de/sat-race-2010`

Instance	#vars	#models (diff)	nH	temps moyen (σ)
12pipe_bug8	117526	10 (1)	0	2,63 (53,32)
ACG-20-10p1	381708	10 (10)	1.42	1452,24 (40,61)
AProVE09-20	33054	10 (10)	33,84	19,5 (9,03)
dated-10-13-s	181082	10 (10)	0,67	6,25 (9,30)
gss-16-s100	31248	10 (1)	0	38,77 (18,75)
gss-19-s100	31435	10 (1)	0	441,75 (35,78)
gss-20-s100	31503	10 (1)	0	681 (58,27)
itox_vc1138	150680	10 (10)	26,62	0,65 (22,99)
md5_47_4	65604	10 (10)	34,8	173,9 (31,03)
md5_48_1	66892	10 (10)	34,76	704,74 (74,65)
md5_48_3	66892	10 (10)	34,16	489,02 (68,96)
safe-30-h30-sat	135786	10 (10)	22,32	0,37 (0,79)
sha0_35_1	48689	10 (10)	33,18	45,4 (21,88)
sha0_35_2	48689	10 (10)	33,25	61,65 (29,93)
sha0_35_3	48689	10 (10)	32,76	72,21 (21,93)
sha0_35_4	48689	10 (10)	33,2	105,8 (35,22)
sha0_36_5	50073	10 (10)	34,19	488,16 (58,58)
sortnet-8-ipc5-h19-sat	361125	4 (4)	15,86	2058,39 (47,5)
total-10-19-s	331631	10 (10)	0,5	5,31 (6,75)
UCG-20-10p1	259258	10 (10)	2,12	768,17 (31,63)
vmpc_27	729	10 (2)	2,53	11,95 (32,62)
vmpc_28	784	10 (2)	3,67	34,61 (25,92)
vmpc_31	961	8 (1)	0	583,36 (88,65)

TABLE 1 – Comportement non déterministe de `ManySAT`

Algorithme 1 : DPLL Deterministe Parallèle

Données : une formule CNF $\mathcal{F}$;
Résultat : $vrai$ si $\mathcal{F}$ est satisfaisable ; $faux$ sinon

1 **début**
2 $<en\ parallele, 0 \leq i < \#coeur>$
3 reponse[i] = search($coeur_i$) ;
4 **pour** *($i = 0$; $i < \#coeur$; i++)* **faire**
5 **si** *($reponse[i]! = inconnu$)* **alors**
6 **retourner** reponse[i];
7 **fin**

même et unique modèle (*diff* =1), la déviation standard du temps d'exécution est d'environ 53 secondes alors que sa moyenne est de 2,63 secondes. Cette première expérimentation illustre clairement à quel point le comportement non déteministe des solveurs SAT parallèles influence la non-reproductibilité des résultats retournés.

4 DPLL Déterministe Parallèle : $(DP)^2LL$

Dans cette partie, nous présentons le premier solveur SAT parallèle et déteministe. Comme le partage de clauses a été prouvé comme étant un élément important dans l'efficacité des techniques de résolution parallèle, notre but est de concevoir une approche déterministe tout en conservant ce partage de clauses. Dans ce but, notre technique de déterminisation est basée sur l'introduction de deux barrières de synchronisation (notées $< barriere >$) qui représentent des points de l'algorithme auxquels les différents fils d'exécution s'attendent les uns les autres. Ces barrières sont placées pour synchroniser à la fois le partage de clauses et la détection de la terminaison de la procédure (cf section 4.1). Par la suite, nous proposons une technique permettant d'ajuster dynamiquement la période de synchronisation avant que les fils ne se rejoignent à la barrière (section 4.2).

4.1 Déterminisation statique

Décrivons maintenant notre approche de déterminisation de solveurs parallèles basés sur un portfolio. Rappelons tout d'abord qu'un solveur basé sur un portfolio exécute différentes instances d'une procédure de type DPLL sur le même problème. Les lignes 2 et 3 de l'algorithme 1 illustrent cet aspect portfolio, en lançant en parallèle plusieurs recherches sur les cœurs de calcul disponibles. Pour éviter tout non-déterminisme en terme de solution retournée ou de preuve d'insatisfaisabilité, une structure de don-

Algorithme 2 : search(coeur$_i$)

Données : une formule CNF $\mathcal{F}$;
Résultat : $reponse[i] = vrai$ si $\mathcal{F}$ est satisfaisable ;
$faux$ si $\mathcal{F}$ is insatisfaisable, $inconnu$ sinon

1 **début**
2 nbConflicts=0;
3 **tant que** *($vrai$)* **faire**
4 **si** *(!propage())* **alors**
5 nbConflicts++;
6 **si** *(topLevel)* **alors**
7 reponse[i]= $faux$;
8 goto $barriere_1$;
9 learntClause=analyse();
10 exporteExtraClause(learntClause);
11 backtrack();
12 **si** *(nbConflicts % periode == 0)* **alors**
13 $barriere_1$: <$barriere$>
14 **si** *($\exists j|reponse[j]! = inconnu$)* **alors**
15 **retourner** reponse[i] ;
16 miseAJourPeriode();
17 importExtraClauses();
18 <$barriere$>
19 **sinon**
20 **si** *(!decide())* **alors**
21 reponse[i]= $vrai$;
22 goto $barriere_1$;
23 **fin**

née globale nommée $reponse$ est utilisée pour enregistrer l'état des différents fils d'exécutions sur la satisfaisabilité de l'instance. Les fils sont triés par rapport à leur identifiant (un entier, de 0 to #coeur-1). L'algorithme 1 retourne le résultat obtenu par le premier cœur (dans cet ordre) si plusieurs d'entre eux peuvent conclure quant à la satisfaisabilité de la formule (lignes 4-6). Ceci est une condition nécessaire mais non suffisante pour la reproductibilité des résultats. Pour parvenir à une déterminisation complète du solveur parallèle, jetons un œil plus attentif au cœur de l'algorithme DPLL alloué à chaque fil (Algorithme 2). En plus des composants habituels des solveurs de type CDCL, on peut observer l'introduction de deux barrières de synchronisation (<*barriere*>, lignes 13 et 18). Pour comprendre le rôle de ces points de synchronisation, nous avons besoin de considérer à la fois leur place au sein de l'algorithme et la région circonscrite par ces deux barrières. Tout d'abord, la première barrière étiquetée $barriere_1$ (ligne 13) est placée avant que tout fil d'exécution puisse retourner l'état de la satisfaisabilité de la formule. Cette barrière les empêche de retourner leur propre solution de manière anarchique, mais les force au contraire à s'attendre entre eux (lignes 14 et 15). Ceci explique pourquoi la procédure est renvoyée à cette barrière dès que l'insatisfaisabilité est prouvée (backtrack au niveau 0 de l'arbre de recherche, lignes 6-8), ou quant un modèle est obtenu (lignes 20-22). A la ligne 14, si la satisfaibilité de la formule est prouvée par l'un des fils ($\exists coeur_j|reponse[j]! = inconnu$), l'algorithme retourne son propre $reponse[i]$. Si aucun ne peut répondre, ils partagent alors de l'information en important les clauses conflits générées par les autres fils durant la dernière *période* (ligne 17). Après que chacun des fils aient fini l'importation (deuxième barrière, ligne 18), ils poursuivent leur exploration, à la recherche d'une solution. Cette seconde barrière est intégrée de manière à empêcher les fils de quitter la région de synchronisation avant les autres. En d'autres termes, quand l'un des fils atteint cette deuxième barrière, il doit attendre que les autres fils aient terminé l'importation des clauses étrangères avant de poursuivre. Comme différents ordres d'importation des clauses peuvent induire différents ordres au niveau de la propagation unitaire, et par conséquent différents arbres de recherche, les clauses apprises sont importées (ligne 17) suivant un ordre fixé par les fils en fonction de leur identifiant.

Pour compléter cette présentation, nous spécifions certaines fonctions usuelles de la procédure de recherche. Tout d'abord, la fonction $propage()$ (ligne 4) applique la propagation unitaire et retourne *faux* si un conflit survient, *vrai* sinon. Dans le premier cas, une clause est apprise par la fonction $analyse()$ (ligne 9), et est ajoutée à la formule puis exportée vers les autres fils (ligne 10, fonction $exporteExtraClause()$). Ces clauses apprises sont périodiquement importées dans la région synchronisée (ligne 17). Dans le cas contraire, la fonction $decision()$ choisit la prochaine variable à affecter et retourne *vrai* ; si elle ne retourne *faux*, alors toutes les variables sont d'ores et déjà affectées et un modèle a été trouvé.

Pour maximiser la dynamique d'échange d'information, chaque fil peut se synchroniser avec les autres avec chaque conflit, en important toute clause apprise juste après sa génération. Malheureusement, comme nous le verrons dans les parties suivantes, cette solution est très inefficace en pratique, puisque beaucoup de temps est perdu par les fils d'exécution à s'attendre entre eux. Pour atténuer ce problème, nous proposons de ne synchroniser les fils qu'après un certain nombre de conflits, noté $periode$ (ligne 10). Cette approche, notée $(DP)^2LL_static(periode)$, ne met pas à jour la période pendant la recherche (pas d'appel à la fonction $miseAJourPeriode()$, ligne 16). Toutefois, bien qu'une valeur optimale puisse être obtenue pour le paramètre $periode$, le problème de l'attente aux barrières ne peut être complètement éliminé. En effet, alors que les différents fils présentent des comportements de recherche différents (stratégie de recherche, taille de la base de clauses apprises, etc.) utiliser la même valeur de $periode$ pour les différents fils conduit inévitablement à gâcher du temps à la première barrière.

Dans la partie suivante, nous proposons une technique de synchronisation dynamique, qui exploite la taille de la formule courante pour déduire une valeur spécifique du paramètre $periode$ pour chaque fil (appel à la fonction $miseAJourPeriode()$, ligne 16).

4.2 Synchronisation dynamique

Comme mentionné plus haut, les procédures de type DPLL exécutées par les différents fils développent différents arbres de recherche. En conséquence, les temps d'attente ne peuvent être évités même si une valeur statique est optimalement définie. Dans cette section, nous proposons une synchronisation dynamique de la valeur du paramètre $periode$. Notre but est de réduire autant que possible le temps gâché en attente à la barrière. Or, le temps nécessaire à chaque fil pour effectuer le même nombre de conflit est difficile à estimer à l'avance. Nous proposons néanmoins une mesure d'approximation, qui estime le coût calculatoire de la propagation unitaire pour en déduire une vitesse de progression de chaque fil. Plus précisément, puisque les différents fils sont exécutés sur la même formule, nous utilisons la taille de la base de clauses apprises courante comme estimation de cette opération basique. Ainsi, notre synchronisation dynamique est calculée en fonction de ces informations.

Décrivons formellement notre approche. Dans notre stratégie de synchronisation, pour chaque unité de calcul u_i, nous considérons une séquence de temps de synchronisation comme une séquence d'étapes t_i^k avec $t_i^0 = 0$ et $t_i^k = t_i^{k-1} + periode_i^k$ où $periode_i^k$ représente la fenêtre de temps définie en terme de nombre de conflits entre t_i^{k-1} et t_i^k. Cette séquence est évidemment différente pour les unités de calculs u_i ($0 \leq i < \#coeur$). Nous définissons Δ_i^k comme l'ensemble de clauses apprises par u_i à l'étape t_i^k. Dans la suite, quand il n'y a aucune ambigüité, nous noterons k plutôt que t_i^k.

Soit $m = max_{\forall u_i}(|\Delta_i^k|)$, où $0 \leq i < \#coeur$ est la taille de la plus grande base de clauses apprises et $S_i^k = \frac{|\Delta_i^k|}{m}$ le ratio entre la taille de clauses apprises de u_i et m. Ce ratio S_i^k représente la vitesse estimée de u_i. Quand ce ratio tend vers 1, la progression de u_i est proche du fil qui est heuristiquement le plus lent, tandis que s'il tend vers 0, le fil u_i progresse beaucoup plus vite que le plus lent. Pour $k = 0$ et pour chaque u_i, nous définissons $periode_i^0$ à α, où α est un entier naturel. Ensuite, à l'étape $k > 0$, et pour chaque u_i, la prochaine valeur est calculée comme suit : $periode_i^{k+1} = \alpha + (1 - S_i^k) \times \alpha$, où $0 \leq i < \#coeur$. Intuitivement, le fil ayant la plus grande valeur pour S_i^k (tendant vers 1) doit avoir la période la plus courte. Au contraire, le fil avec le plus petit S_i^k (tendant vers 0) doit avoir la période la plus longue.

5 Évaluation expérimentale

Toutes nos expérimentations ont été conduites sur des processeurs Intel Xeon 3GHz sous Linux CentOS 4.1. (noyau 2.6.9) avec une limite de mémoire vive de 2 Go. Notre algorithme DPLL déteministe a été implanté sur le solveur parallèle basé un portfolio `ManySAT` (version 1.1). Le temps limite a été fixé à 900 secondes pour chaque instance, et si aucune réponse n'a été apportée dans cette limite de temps, l'instance est considérée non résolue. Nous avons utilisé les 100 instances proposées lors de la récente `SAT Race 2010`, et nous reportons pour chaque expérimentation le nombre d'instances résolues (x-axis) avec le temps nécessaire à leur résolution (y-axis). Chaque solveur parallèle est exécuté en utilisant quatre fils d'exécution. Notons que dans les expérimentations suivantes, plutôt que le temps CPU utilisé habituellement, nous avons préféré considéré le temps réel. En effet, dans la plupart des architectures, le temps CPU n'est pas incrémenté quand les fils d'exécution sont endormis (temps d'attente à la barrière). Ainsi, prendre en considération le temps CPU donnerait à nos implantations déterministes un avantage à la fois substantiel et illégitime.

5.1 Période statique

Dans cette première expérimentation, nous avons évalué les performances de notre algorithme **DPLL D**éterministe et **P**arallèle ($(DP)^2$LL) avec différentes périodes de synchronisation. La Figure 1 présente les résultats obtenus. Tout d'abord, une version séquentielle du solveur a été exécutée. Sans surprise, cette version obtient les pires résultats globaux en ne résolvant que 68 instances en plus de 11 000 secondes. Ce résultat permet d'apprécier le gain d'obtenu par les mécanismes parallèles. Nous reportons également les résultats obtenus par l'exécution de la version non déterministe du solveur `ManySAT 1.1`. Soulignons que, comme présenté dans la section 3, lancer plusieurs fois ce solveur peut conduire à des résultats différents. Ce solveur non déterministe a permis la résolution de 75 instances en 8 850 secondes. Ensuite, nous avons exécuté une version déteministe de ce même solveur, dans laquelle chaque fil se synchronise après la génération de chaque clause conflit ((DP)2LL_static(1)). Il est possible d'observer le poids conséquent, d'un point de vue calculatoire, de cette barrière. En effet, cette version est clairement moins efficace que la version non déterministe, en ne résolvant que 72 instances sur les 100 proposée, et cela en plus de 10 000 secondes.

Ce résultat négatif est principalement dû au temps perdu par les fils d'exécution à s'attendre les uns les autres de manière très fréquente. Pour pallier ce problème, nous avons également essayé de ne synchroniser les fils qu'après un certain nombre de conflits (100, 10 000). La Figure 1 montre que ces versions surpassent en efficacité la version

periode	**temps de résolution**	**temps d'attente**	**attente/résolution (en %)**
static(1)	10 276	4 208	40,9
static(100)	9 715	2 559	26,3
static(10000)	9 124	1 605	17,6

TABLE 2 – Temps d'attente aux barrières de synchronisation par rapport à la période

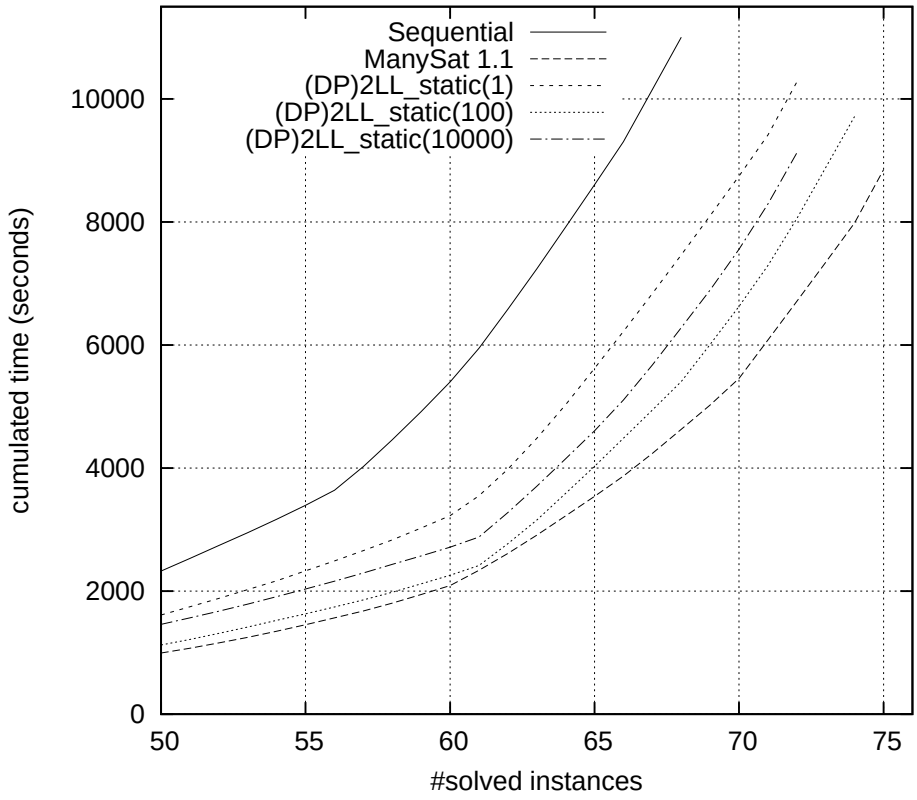

FIGURE 1 – Performances avec synchronisation statique

« periode=1 », mais elles restent loin des résultats obtenus avec la version non déterministe.

Ces deux barrières permettant à notre algorithme d'être déterministe sont clairement un inconvénient à son efficacité. Ainsi, nous avons évalué la quantité de temps passé par les fils d'exécution à s'attendre les uns les autres. Dans la table 2, nour fournissons le rapport de temps passé à attendre dans un but de synchronisation sur le temps d'exécution total pour les différentes versions de `(DP)2LL`. Comme les résultats le montrent, ce temps peut être très significatif quant l'attente est faite après chaque conflit. En effet, dans ce cas, plus de 40% du temps total est passé par les fils d'exécution à s'attendre entre eux. Comme nous le prévoyions, ce temps est réduit quand la synchronisation ne survient que plus épisodiquement, mais n'est jamais négligeable. Notons que bien que la version « periode=10000 » perd moins de temps à attendre que la version« periode=100 », elle obtient les pires résultats globaux en ne résolvant que 72 instances. La version « periode=100 » obtient les meilleurs résultats (si on exclut la version non déterministe). Après des expérimentations intensives, ce paramètre semble être un bon compromis entre le temps d'attente à la barrière et la dynamique d'échange d'informations. En effet, nous pensons que pour la version « periode=10000 », l'information est échangée trop tard, ce qui explique son comportement peu satisfaisant.

De ce fait, nous avons tenté de réduire le temps que les fils d'exécution passent à attendre à la barrière en adoptant une stratégie dynamique (présentée dans section 4.2). Les résultats empiriques concernant cette version sont présentées dans la partie suivante.

5.2 Période dynamique

Dans une seconde expérimentation, nous avons evalué notre stratégie dynamique. Nous comparons les résultats de cette version contre ceux obtenus par `ManySAT 1.1`, et

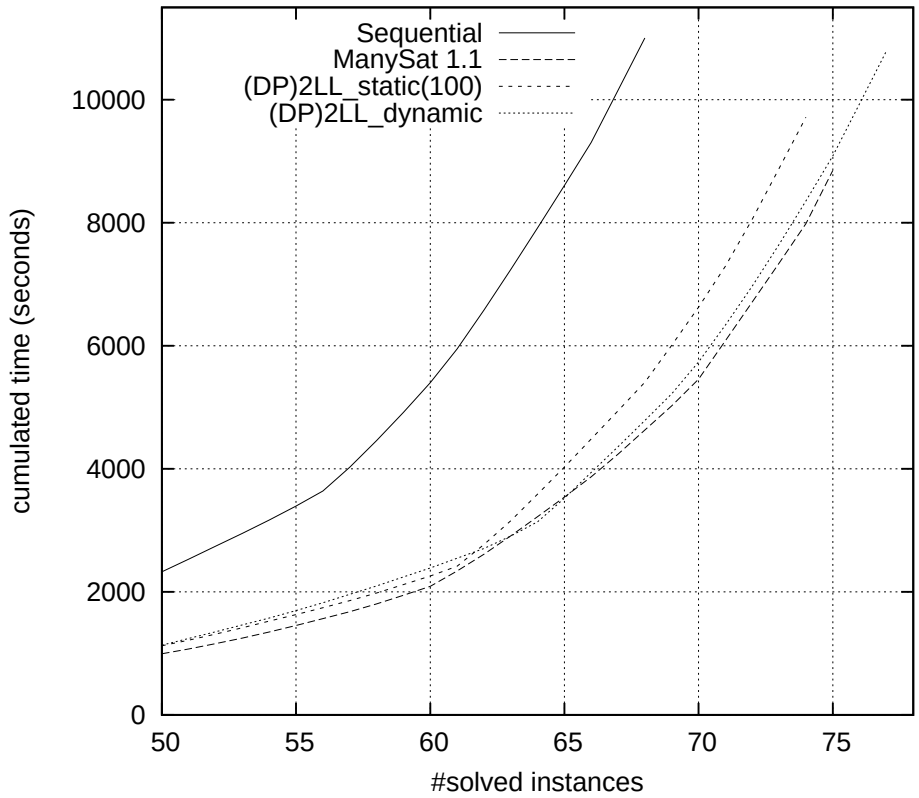

FIGURE 2 – Performances avec synchronisation dynamique

fournissons également les résultats de la meilleure version statique (100), et de la version séquentielle. Les résultats obtenus sont reportées dans la Figure 2. Cette version dynamique est exécutée avec le paramètre $\alpha = 100$.

Cette expérimentation confirme l'intuition suivant laquelle chaque fil d'exécution doit avoir une période de synchronisation différente, suivant la taille de leur base de clauses apprises, permettant d'estimer heuristiquement sa vitesse de propagation. En effet, on peut observer dans la Figure 2 que la « courbe de résolution » de cette version dynamique est très proche de celle `ManySat 1.1`. Ceci montre que les 2 solveurs sont capables de résoudre à peu près le même nombre d'instance dans des temps similaires. De plus, cette version adaptative est capable de résoudre 2 instances de plus que la version non déterministe, ce qui en fait la version la plus efficace testée pendant nos expérimentations.

6 Discussion

Dans cet article, nous apportons une première réponse au problème important du non-déteminisme que pose les solveurs SAT parallèles, en proposant $(DP)^2LL$ une première implantation d'une procédure déterministe parallèle. A cette fin, une idée simple mais efficace est présentée : elle consiste principalement en l'introduction de barrières de synchronisation à l'algorithme originel. Nous proposons plusieurs stratégies de synchronisation, et montrons que cette version déterministe se montre empiriquement très efficace. En effet, ses résultats sont similaires à une version non déterministe.

Cette première implantation ouvre de nombreuses perspectives de recherche. Tout d'abord, comme dans la plupart des procédures parallèles basées sur l'échange de clauses, le nombre d'unités de calcul est un goulot d'étranglement pour l'efficacité de la recherche. Exécuter des solveurs sur un grand nombre d'unités de calcul (par exemple 32) ne garantit pas que de meilleurs résultats seront obtenus. En fait, comme les clauses sont échangés par un plus grand nombre de travailleurs, les techniques actuelles sont souvent moins efficaces quand elles sont utilisées sur un grand nombre d'unités de calculs. Dans ce papier, nous avons proposé une approche garantissant le déterminisme aux solveurs SAT parallèles. Une autre caractéristique clé à ajouter à ces solveurs serait l'opportunisme par rapport aux ressources utilisées (nombre d'unités de calcul, mémoire, etc.). Nous prévoyons d'étudier cela dans un travail futur. Ensuite, les barrières ajoutées à la boucle principale de l'algorithme CDCL parallèle peuvent être vues comme un revers à l'efficacité pratique de la procédure. En effet, chaque fil d'exécution ayant terminé son calcul partiel doit attendre que les autres fils aient également terminé. Néanmoins, notre

politique de synchronisation se montre très efficace tout en conservant le cœur de l'architecture parallèle : l'échange de clauses. De nouvelles techniques permettant aux fils d'exécution d'interagir peuvent être imaginées à cet endroit particulier de la recherche. Nous pensons que ceci est une perspective de recherche très intéressante.

Références

[1] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT*, pages 28–33, 2008.

[2] A. Biere. Lazy hyper binary resolution. Technical report, Dagstuhl Seminar 09461, 2009.

[3] A. Biere. Lingeling, plingeling, picosat and precosat at SAT race 2010. Technical Report 10/1, FMV Reports Series, 2010.

[4] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4) :381–400, 1996.

[6] W. Chrabakh and R. Wolski. GrADSAT : A parallel SAT solver for the grid. Technical report, UCSB Computer Science, 2003.

[7] G. Chu and P. J. Stuckey. Pminisat : a parallelization of minisat 2.0. Technical report, SAT Race, 2008.

[8] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *15th International Conference on Principles and Practice of Constraint Programming - CP 2009*, pages 226–241, 2009.

[9] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[10] Luis Gil, Paulo Flores, and Luis Miguel Silveira. PMSat : a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :71–98, 2008.

[11] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

[12] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT : a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262, 2009.

[13] S. Kottler. SAT solving with reference points. In *SAT*, pages 143–157, 2010.

[14] S. Kottler. SArTagnan : solver description. Technical report, SAT Race 2010, July 2010.

[15] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

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

[17] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3) :363–382, 2009.

[18] Carl Christian Rolf and Krzysztof Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 304–309, 2008.

[19] T. Schubert, M. Lewis, and B. Becker. Antom : solver description. Technical report, SAT Race, 2010.

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