SwarmSAT: un solveur SAT massivement parallèle

Jean-Marie Lagniez **Nicolas Szczepanski** **Sébastien Tabary**
Univ. Lille-Nord de France, CRIL/CNRS UMR8188, Lens, F-62307 CRIL-CNRS

{lagniez,szczepanski,tabary}@cril.fr

Résumé

Dans ce papier, nous discutons de l'efficacité de la résolution du problème SAT dans un cadre massivement parallèle. Plus précisément, nous proposons une première version d'un solveur SAT, nommé SwarmSAT, fondé sur une approche de type collaborative à la CubeAndConquer et où la partie communication est gérée grâce à la bibliothèque Open MPI. Nous montrons expérimentalement que l'approche de base, qui consiste à diviser l'espace de recherche sous forme de cubes et de les résoudre en parallèle, n'est pas viable en pratique. Ces explications mettront en évidence les axes d'améliorations à apporter à SwarmSAT pour de futurs travaux.

1 Introduction

Longtemps étudié dans un contexte séquentiel pour ses nombreuses applications industrielles (planification, vérification formelle, ...), le problème SAT a fait éclore des algorithmes efficaces et puissants qui s'expliquent à la fois par l'amélioration des solveurs SAT modernes (apprentissage, VSIDS, *watched literal*, ...) et l'accroissement de la puissance de calcul. Aujourd'hui, les solveurs SAT permettent la résolution de problèmes de taille importante contenant des millions de variables et de clauses. Néanmoins, certaines instances restent encore inabordables en raison, en autres, de leur structure. La parallélisation des algorithmes de résolution pour SAT constitue un axe d'amélioration. Ceci est renforcé par le fait qu'à l'heure actuelle la puissance de calcul d'un ordinateur ne se traduit plus par une augmentation des fréquences du microprocesseur, mais par l'augmentation du nombre de cœurs de calcul au sein de celui-ci. Dans ce contexte, nous distinguons deux modèles pour la résolution parallèle du problème SAT : le modèle concurrentiel qui exécute en parallèle plusieurs solveurs séquentiels en concurrence sur le même problème (`Plingeling` [6], `PeneLoPe` [1], ...) ; et le modèle collaboratif (coopératif) fondé sur le paradigme « diviser pour mieux régner » (`CubeAndConquer` [11], `Dolius` [4], ...).

Dans ce papier, nous présentons un solveur modulaire collaboratif, nommé `SwarmSAT`, basé sur une résolution massivement parallèle de cubes générés selon la méthode proposée dans `CubeAndConquer` [11]. Les communications entre les différentes unités de calcul sont gérées grâce à la bibliothèque de communication Open MPI. Dans la partie expérimentation, nous étudions l'impact des différents composants de `SwarmSAT` (qualité des cubes, coût des communications, ...). Puis, nous comparons `SwarmSAT` avec un solveur collaboratif différent nommé `Dolius` [4] et nous discutons des axes d'améliorations possibles.

2 Préliminaire

Nous supposons le lecteur habitué avec les concepts de variables, de clauses et d'interprétations usités dans le contexte de SAT (voir [7], pour plus d'informations). Dans cette section nous nous focalisons uniquement sur la description du solveur collaboratif `CubeAndConquer`[8].

Le solveur `CubeAndConquer`, initialement proposé pour être exécuté sur une grille de calcul, consiste à diviser l'espace de recherche en millions de cubes (un cube (ou terme) est une conjonction de littéraux) et à les résoudre en parallèle. Étant donné l'importance des cubes, les auteurs proposent d'utiliser un solveur *look-ahead* de type DPLL pour la génération de ces derniers. Cette approche développe un arbre de recherche dans lequel chaque branche correspond à un cube. La profondeur de cet arbre est bornée par une heuristique. Une fois la génération des cubes terminée, ces derniers sont résolus par un ensemble de solveurs `Plingeling` (de type CDCL) en parallèle.

3 SwarmSAT

Dans cette section, nous décrivons les trois types de processus utilisés dans le solveur `SwarmSAT`, ainsi que la manière dont les communications entre ces derniers sont réalisées.

3.1 Les différents processus

Le scout : Il se charge de la construction des cubes de la même manière que `CubeAndConquer` (ie. utilisation du solveur `march_cc` [12]).

Le worker : C'est un solveur SAT qui se charge de la résolution des cubes. Afin de profiter du travail déjà réalisé et d'obtenir une raison lors de l'échec de la résolution de ce dernier, le cube sera passé en *assumption* et le problème résolu de manière incrémentale (voir [3] pour plus d'informations). Ainsi, le résultat retourné par le solveur peut être de deux types : soit il répond SAT et le problème est donc montré globalement satisfaisable ; soit le cube est réfuté. Dans ce dernier cas, il est possible, grâce à la notion d'*assumption*, d'extraire une raison de l'insatisfaisabilité du cube. Si la clause ainsi générée est vide, l'insatisfaisabilité est donc montré globalement, sinon l'insatisfaisabilité est locale et le *worker* demande un nouveau cube à résoudre.

Le master : Il se charge de l'initialisation et de la communication entre les *workers* et le *scout*. C'est ce dernier qui va s'occuper de gérer les cubes produits par le *scout* (ils sont stockés dans une file) et qui les distribue au *worker* en attente. Le *master* a aussi la tâche de centraliser les clauses provenant des cubes déjà prouvés insatisfaisables. Grâce à ces dernières, il a également la possibilité de ne pas considérer un cube qui contredit une des clauses.

3.2 Communication dans `SwarmSAT`

Les communications sont réalisées point-à-point via des messages non-bloquants. Elles sont utilisées pour la transmission des cubes ainsi que le passage de certaines clauses des *workers* au *master* ou encore pour la transmission de la solution. Remarquons que dans la version actuelle de `SwarmSAT`, aucune des clauses apprises n'est échangée entre les *workers*.

4 Mode de fonctionnement

`SwarmSAT` est un solveur modulaire qui permet d'ajouter facilement de nouveaux types de *workers*, de gérer une création de cubes dynamique ou statique et aussi de résoudre une instance du problème SAT en parallèle que ce soit en mode collaboratif (à la `CubeAndConquer`) ou concurrentiel (à la `ppfolio` [13]).

La gestion de l'ajout d'un nouveau solveur est ainsi réalisée grâce à l' implémentation d'une interface écrite en C++. Elle permet de définir les méthodes nécessaires à la gestion des communications via Open MPI. Dans la version expérimentée de ce papier, nous avons considéré les solveurs `Glucose` [5] et `MiniSat` [10] pour implémenter les *workers*.

En ce qui concerne le scout, nous utilisons la version de `march_cc` fournie par les auteurs de `CubeAndConquer`. Cependant, nous avons aussi proposé une version dynamique qui, contrairement à la version initiale, permet de rendre disponibles les cubes durant leur génération (dans `CubeAndConquer` il fallait parfois attendre plus de 20 minutes pour obtenir tous les cubes). Cela nous permet d'anticiper la résolution des cubes même si en contrepartie, certains cubes auraient été prouvés insatisfaisables durant la phase de création.

Dans la suite, nous nommerons $\texttt{SwarmSAT}(\texttt{W}, \texttt{S}, \texttt{G})_{\texttt{C}}$ pour signifier que nous utilisons comme *worker* des solveurs de type `W` (`Glucose`, `MiniSat`, ...) et un *scout* utilisant la méthode `S` pour la génération des cubes. La lettre `G` peut prendre comme valeur `dynamic` ou `static` en fonction de la disponibilité des cubes (directement ou après leur création) tandis que `C` permet de connaître le nombre de workers utilisés.

5 Expérimentations

L'ensemble des expérimentations ont été réalisées sur 64 cœurs grâce à deux nœuds de 32 cœurs. Les nœuds de calcul sont des Dell R910 avec 4 Intel Xeon X7550 contenant chacun huit cœurs. Chaque unité de calcul dispose d'un contrôleur Gigabit Ethernet et de 256 Go de RAM.

Les expérimentations présentées dans ce qui suit utilisent le même protocole (instances et temps de résolution) que celui proposé par les auteurs de `Dolius` [4]. Plus précisément, nous considérons 20 minutes comme limite de temps réel. Les instances sélectionnées sont issues de la compétition SAT 2013 (*application track*) [2]. Une instance est sélectionnée si elle a été résolue par au moins un des cinq premiers solveurs parallèles de la compétition et ne peut être résolue par tous ces solveurs. Cela permet d'avoir des instances difficiles et diversifiées : 18 satisfaisables et 33 insatisfaisables.

Dans `SwarmSAT`, étant donné la nature du *scout* (durée limitée) et du *master* (peu de travail), nous avons choisi de faire cohabiter ces derniers avec les *workers*.

Nous comparons le solveur `Dolius` [4] avec différentes versions de `SwarmSAT` présentées subséquemment :

- S(G,∅,∅)$_{64}$ pour SwarmSAT(Glucose$_{rdm}$, ∅, ∅)$_{64}$: la génération des cubes étant vide, cette version est un solveur concurrentiel. Afin que les solveurs rentrent en concurrence nous utilisons une version de Glucose avec une graine aléatoire différente pour chaque *worker* ;
- S(G,m,d)$_{64}$ pour SwarmSAT(Glucose, march_cc, dynamic)$_{64}$: nous utilisons 64 *workers* de type Glucose et march_cc en mode dynamique pour la génération des cubes ;
- S(M,m,d)$_{64}$ pour SwarmSAT(MiniSat, march_cc, dynamic)$_{64}$: nous utilisons 64 *workers* de type MiniSat et march_cc en mode dynamique pour la génération des cubes ;
- S(G,m,s)$_{64}$ pour SwarmSAT(Glucose, march_cc, static)$_{64}$: nous utilisons comme *workers* 64 solveurs Glucose et march_cc en mode statique. Ici, les *workers* tentent de résoudre l'instance en deux étapes. Dans un premier temps, sans cube durant le temps nécessaire à l'exécution du *scout*. Puis dans un seconds temps, via les cubes.

Solveur	SAT(18)	UNSAT (33)	Total (51)
S(G,∅,∅)$_{64}$	12	12	24
S(G,m,s)$_{64}$	10	11	21
S(M,m,d)$_{64}$	10	1	11
S(G,m,d)$_{64}$	6	4	10
Dolius	**13**	**24**	**37**

TABLE 1 – Résultats des différentes versions de SwarmSAT et de Dolius

Le tableau 1 reporte les résultats obtenus par les différentes approches expérimentées dans ce papier. Tout d'abord, nous discutons des résultats obtenus par le solveur Dolius. Nous pouvons remarquer que ce dernier est significativement meilleur sur les instances insatisfaisables (24 instances insatisfaisables résolues) que les différentes versions de SwarmSAT testées (12 pour la meilleure version de SwarmSAT). Cela s'explique largement par le fait que contrairement à SwarmSAT, Dolius permet d'échanger les clauses apprises entre les différentes unités de calcul. Cet échange d'informations permet de guider les solveurs plus rapidement vers la génération d'une clause vide. Pour ce qui est des instances satisfaisables, nous pouvons voir que l'impact de l'échange d'informations n'est pas aussi prépondérant et qu'une version concurrentielle simple telle que S(G,∅,∅)$_{64}$ permet de résoudre approximativement le même nombre d'instances (13 pour Dolius et 12 pour S(G,∅,∅)$_{64}$).

Concernant les résultats obtenus par les différentes versions de SwarmSAT, nous observons qu'une division de l'espace de recherche à l'aide de cubes, telle qu'elle a été proposée dans [11], n'est pas efficace si elle n'est pas accompagnée d'un échange d'informations

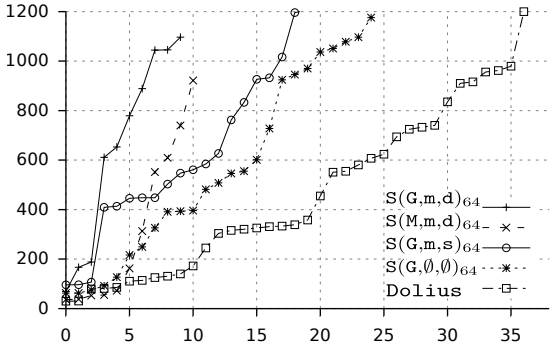

FIGURE 1 – Nombre d'instances résolues en fonction du temps des SwarmSATs et Dolius.

entre les solveurs. En effet, la version concurrentielle de SwarmSAT S(G,∅,∅)$_{64}$, qui ne divise pas l'espace de recherche à l'aide de cubes, fournie les meilleurs résultats (12 instances satisfaisables résolues et 12 instances insatisfaisables résolues). De plus, comme le montre la figure 1, S(G,∅,∅)$_{64}$ résout aussi plus rapidement les instances. Afin de vérifier que la perte de performances de l'approche basée sur les cubes n'était pas due au coût de communication, nous avons mesuré ces derniers. Il s'avère que le temps de transfert des cubes du *master* aux *workers* ainsi que le passage des clauses apprises des *workers* au *master* est insignifiant (moins de 2 secondes au total).

Les expérimentations ne permettent pas de conclure sur la qualité des cubes. En effet, bien que S(G,m,s)$_{64}$ résolve significativement plus d'instances (10 instances satisfaisables et 11 instances insatisfaisables) que S(G,m,d)$_{64}$ (6 instances satisfaisables et 4 instances insatisfaisables) et S(M,m,d)$_{64}$ (10 instances satisfiables et 1 instance insatisfaisable), dans la plupart des cas cela est une conséquence du fait que la génération de tous les cubes est trop coûteuse en temps. Pour 33 instances la génération ne termine pas et pour les 18 instances restantes, cette dernière nécessite en moyenne 296.27 secondes du temps alloué à la résolution. Ces chiffres montrent que dans la majeure partie du temps S(G,m,s)$_{64}$ est équivalent à S(G,∅,∅)$_{64}$. Cela explique largement les performances de S(G,m,s)$_{64}$.

Finalement, nous pouvons voir que le choix du solveur utilisé pour implémenter les *workers* est prépondérant. Ainsi, le choix du solveur MiniSat permet de résoudre plus facilement les instances satisfaisables tandis qu'utiliser Glucose permet d'être plus efficace sur les instances insatisfaisables. Ces résultats s'expliquent par le fait que cette différence de performance en fonction de la satisfaisabilité de l'instance existe déjà entre ces deux solveurs lorsqu'ils sont exécutés séquentiellement. Néanmoins, il semble que dans le cas d'une résolution incrémentale le phénomène est accentué.

6 Conclusion et perspectives

Ce papier présente un nouveau solveur SAT, nommé `SwarmSAT`, largement inspiré de `CubeAndConquer` et dédié au massivement parallèle. Cette première mouture utilise pour la génération des cubes le solveur `march_cc` [11] et comme *workers* l'un des deux solveurs de l'état de l'art `Glucose` et `MiniSat`. Les résultats expérimentaux montrent que dans le cadre de la résolution d'instances industrielles `SwarmSAT` (plus généralement, les approches fondées sur `CubeAndConquer`) ne sont pas efficaces en pratique (sauf dans le cas où les instances seraient de type académique).

Bien que les résultats obtenus dans ce papier ne soient pas à la hauteur de nos attentes, les travaux réalisés autour de la création du solveur `SwarmSAT` forment une base solide pour de nombreuses extensions. En effet, nous avons la possibilité d'actionner plusieurs leviers afin d'améliorer notre approche.

En ce qui concerne la génération des cubes, de nombreuses pistes restent à explorer. Dans un premier temps, nous envisageons de modifier le solveur `march_cc` afin de contrôler le temps nécessaire à la création des cubes. Par exemple, nous pouvons réduire la profondeur de l'arbre de recherche exploré. Nous examinons aussi la possibilité d'itérer le processus de génération de cubes. Plus précisément, nous pourrions dans un premier temps générer des cubes avec une petite profondeur d'arbre afin de fournir rapidement des cubes aux *workers* et de les « améliorer » en augmentant cette dernière. Il serait aussi possible de régénérer un nouveau pool de cubes si ces derniers sont considérés mauvais par les *workers* (par exemple si les solveurs n'avancent pas dans la résolution des cubes).

Un fait marquant des expérimentations est que pour beaucoup d'instances le nombre de cubes résolus est insignifiant (par exemple pour `bob12s06` seulement 4% des cubes ont été traités). Une telle situation peut conduire à focaliser la recherche sur une seule partie de l'espace de recherche. Cela peut être très préjudiciable lorsque nous observons l'impact des redémarrages dans les solveurs SAT modernes. Afin d'atténuer ce phénomène, nous envisageons de faire en sorte que des cubes puissent obtenir le statut indéterminé. De cette manière un *worker* peut libérer un cube et en choisir un autre. Par la suite, le cube ainsi libéré pourra être retravaillé.

Une autre piste d'amélioration concerne l'échange d'informations entre les *workers*. Notons que cela est d'une certaine manière déjà réalisée via le transfert des clauses apprises lorsqu'un cube est réfuté. Il est tout à fait envisageable de partager des clauses entre les différentes unités de calcul. Cependant, souhaitant rendre applicable `SwarmSAT` dans le cadre massivement parallèle, une attention particulière devra être apportée à la manière donc cet échange sera réalisé. Pour cela, nous pourrons nous inspirer des travaux présentés dans [1] (pour le choix des clauses à partager) et [9] (pour réaliser cet échange).

Finalement, au vu des résultats obtenus par $\texttt{S(G,m,d)}_{64}$ et $\texttt{S(M,m,d)}_{64}$, il semblerait qu'il y ait fort à gagner à utiliser plusieurs types de solveurs pour implémenter les *workers*.

Références

[1] Gilles A., B. Hoessen, S. Jabbour, J-M. Lagniez, and C. Piette. PeneLoPe, a Parallel Clause-Freezer Solver. In *SAT'12*, pages 43–44, 2012.

[2] A. Belov A. Balint, M. Heule and M. Jarvisalo. The application and the hard combinatorial benchmarks in sat competition 2013. In *SAT'13*, page 99, 2013.

[3] G. Audemard, A. Biere, J-M. Lagniez, and L. Simon. Améliorer SAT dans le cadre incrémental. *RIA*, pages 593–614, 2014.

[4] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette. Dolius : A distributed parallel sat solving framework. In *POS'14*, 2014.

[5] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI'09*, pages 399–404, 2009.

[6] A. Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. In *SAT'13 : Solver and Benchmarks Descriptions*, page 51, 2013.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, 2009.

[8] C. Biró, G. Kovásznai, A. Biere, G. Kusper, and G. Geda. Cube-and-conquer approach for sat solving on grids. *Annales Mathematicae et Informaticae*, pages 9–21, 2013.

[9] T. Ehlers, D. Nowotka, and P. Sieweck. Communication in massively-parallel sat solving. In *ICTAI'14*, pages 709–716, 2014.

[10] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT'03*, pages 502–518, 2003.

[11] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer guiding CDCL SAT solvers by lookaheads. In *HVC'11*, pages 50–65, 2011.

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

[13] Olivier Roussel. Description of ppfolio. 2011.