Un nouveau cadre diviser pour régner pour SAT distribué

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Cédric Piette *

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

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

Résumé

Nous proposons dans cet article une nouvelle technique parallèle pour SAT. Celle-ci, basée sur le paradigme "diviser pour régner", met en place une architecture distribuée de type maître/esclave, où chaque esclave est un solveur CDCL travaillant sur une sous-formule du problème initial, tandis que le maître est chargé de mettre en relation les esclaves, et ne travaille donc pas "activement" à la résolution. Le schéma de répartition de charge adopté est celui du vol de travail, dont la division peut être effectuée "classiquement" par séparation sur une variable, mais autorise également des formes plus sophistiquées de division. Notre implémentation préliminaire permet d'obtenir des résultats très honorables.

Abstract

In this paper, we propose a new distributed approach for SAT. This approach, based on the divide-and-conquer principles, relies on master/slave architecture, where each slave is a CDCL solver working on a subformula of the initial problem, whereas the master does not actively participate to the solving process, since its main task is to coordonate the slaves, and etasblishing communication between them. The load balancing strategy is the work stealing technique, where the load division can be achieved in a classical way by splitting on variables. However, our division scheme allows more sophisticated splitting forms. Our preliminary experiments delivers very satisfying results.

1 Introduction

Le problème SAT, qui consiste à décider de la satisfaisabilité d'une formule booléenne sous forme normale conjonctive, occupe un rôle central en théorie de la complexité, où il représente le problème NP-complet de référence [6], alors que de nombreux problèmes issus d'applications pratiques (vérification formelle, cryptographie, bio-informatique, etc.) s'y ramènent naturellement ou le contiennent.

Il est le sujet, depuis des décennies, de recherches actives concernant sa résolution pratique. De très nombreux leviers (structure de données, heuristique, apprentissage, etc.) ont été mis en évidence pour réduire encore et toujours l'effort calculatoire nécessaire à déterminer la satisfaisabilité d'un problème donné. Ces dernières années, l'évolution matérielle des unités de calcul a principalement consisté en une multiplication de cœur de calculs. Ainsi, depuis quelques années, on voit émerger un intérêt croissant pour la résolution parallèle de SAT.

Dans ce papier, nous présentons un cadre (appelé `Dolius`) permettant de résoudre SAT de manière distribuée, selon le principe du diviser pour régner. Ce travail préliminaire représente la première étape d'un projet à plus large échelle. Dans cette approche, le maître est chargé de mettre en relation les esclaves et ne travaille donc pas activement. Lorsqu'un esclave a terminé son travail, il peut répondre SAT et mettre un terme à la recherche globale, soit répondre UNSAT et dans ce cas demander du travail au maître. Le maître le met alors en relation avec un autre esclave, celui-ci va alors diviser son travail en deux permettant au premier esclave d'effectuer une nouvelle recherche. Il existe différentes possibilités pour diviser la recherche. La séparation sur une variable est la solution classique. Une autre est la séparation du travail en utilisant deux ensembles de clauses mutuellement inconsistants [12]. Si tous les esclaves ont répondu UNSAT alors le pro-

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

blème initial est lui même UNSAT. Notre approche distribuée utilise les protocoles réseau et peut donc mettre en relation un nombre quelconque d'esclaves.

Le reste de cet article est structuré de la manière suivante : la section 2 introduit les définitions et notations nécessaires à la compréhension, la section 3 passe en revue les différentes approches existantes pour paralléliser la résolution du problème SAT. La section 4 présente notre nouvelle approche. Enfin, cet article se termine par une première évaluation expérimentale de `Dolius` + `PeneLoPe`, notre solveur SAT distribué.

2 Définitions et notations

Le lecteur est supposé familier avec les notions liées au problème de satisfaisabilité d'une formule propositionnelle (variable x, littéral positif x ou négatif $\neg x$, clause, clause unitaire, interprétation et CNF). Chaque esclave de notre approche utilise un solveur de type CDCL (*Conflict-Driven, Clause Learning*) dont voici une brève explication du fonctionnement. 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, 20]. 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 cet article, la distinction est faite entre une application parallèle et une application distribuée. La première utilise un système de mémoire centrale pour la communication inter processus, tandis que la seconde communique au travers d'interface(s) réseau. L'avantage d'un système à mémoire centrale est la rapidité des communications. Les applications distribuées permettent en revanche de faciliter l'utilisation d'un plus grand nombre d'unités de calcul par des machines distinctes.

Dans le cadre d'applications multi-processus, un processus est dit *affamé* si celui-ci est disponible mais n'a pas de travail à effectuer.

3 État de l'art

Il existe deux approches générales pour paralléliser la résolution SAT. D'un côté, les algorithmes de type *portofolio* consistent à placer en concurrence plusieurs instances d'un solveur, chacune d'entre elles étant paramétrées différemment. L'idée de cette approche est d'utiliser des stratégies orthogonales afin de raccourcir en pratique le temps global de résolution, toutes les instances du solveur étant stoppées dès l'obtention de la solution par la stratégie la plus adaptée au problème traité.

Toutefois, certains portofolios de solveurs ne sont pas uniquement basés sur la concurrence. Certains mettent également en place des techniques de partage d'information. En pratique, ce partage est basé sur l'échange de clauses apprises au cours de la recherche (*nogood*). À titre d'exemple, chaque instance du solveur `plingeling` [4] transmet les clauses unitaires produites au cours de la recherche, afin de faciliter les autres recherches. D'autre mécanismes plus sophistiqués ont été récemment proposés, tels que ceux introduits dans le portofolio `ManySAT` [9]. Toutefois, le partage d'informations au sein d'un portofolio est délicat. D'un côté, certaines informations obtenues par une instance de solveur peuvent être cruciales à la recherche d'une de ces concurrentes, et accélérer considérablement l'exploration de l'espace de recherche. Il est donc souhaitable de transmettre ces informations. D'un autre côté, la surabondance d'informations, et le coût de leur transmission d'une instance aux autres peuvent véritablement pénaliser l'efficacité du portofolio. Ce problème est accru par le fait (i) qu'un nogood est appris après chaque conflit rencontré par chacune des instances (ii) il est extrêmement difficile de prédire l'utilité d'un *nogood*. Pour pallier ces difficultés, `PeneLoPe` [1] se base sur différents concepts heuristiques tels que le *Progress Saving Measure* (`PSM`) [2] et le *Literal Block Distance* (`LBD`) [3] qui ont pour objectif de ne partager que les clauses jugées les plus pertinentes pour les recherches en cours.

L'autre approche majeure pour paralléliser SAT se fonde sur l'idée de diviser pour régner. Cette classe d'algorithmes a pour principe de réduire récursivement un problème en un ou plusieurs sous-problèmes du même type, plus simple à résoudre. Appliqué à SAT, cette technique vise à diviser la formule à résoudre Σ en plusieurs sous-formules $\Sigma_1, \Sigma_2, ...$ qui sont ensuite résolues indépendamment par différentes unités de calcul. Il s'agit de l'un des aspects importants de ce modèle, qui réside dans le partage de la charge de travail entre les processeurs.

Le projet `SAT@HOME` [16], est un exemple de parallélisation de type diviser pour régner. Celui-ci, initié en 2010, se base sur la plate-forme de calcul distribué `BOINC` [17] pour effectuer une résolution massivement parallèle, de type diviser pour régner. Ces travaux héritent de l'architecture du célèbre projet `SETI@HOME`,

visant à distribuer l'observation et l'analyse des rayons issus de l'espace, dans le but de détecter de la vie intelligente non terrestre.

Il existe également d'autres implémentations de type diviser pour régner pour résoudre SAT. Par exemple, Feldman *et al.* [8] proposent une telle architecture, qui permet en outre le partage des clauses apprises (*no-good*) entre les différents processus. Cependant, ce partage de clauses se fait au sein d'une zone mémoire commune à différents fils d'exécution, ce qui rend cette implémentation impossible à exécuter sur des machines distinctes.

On compte aussi parmi les approches distribuées pour résoudre SAT le portail `GridSat` [5]. Lancé en 2000, il a pour but de fournir une interface publique simple à utiliser pour exécuter une résolution distribuée à large échelle (c'est à dire sur un grand nombre de calculateurs). Cet ambitieuse plate-forme est malheureusement ancienne et n'exploite pas les dernières avancées algorithmiques pour la résolution de SAT. Elle est en effet basée sur `Chaff`, le solveur connu pour avoir introduit les *watched littéraux*. Ce solveur n'est toutefois plus maintenu et n'est plus représentatif de l'état de l'art pour la résolution de SAT.

Récemment, Schulz et Blochinger [18] ont proposé une approche originale pour résoudre SAT de manière distribuée. En effet, contrairement à la plupart des autres techniques distribuées, où un élément centralisateur (souvent appelé *maître*) est nécessaire, leur contribution consiste à proposer un système *pair-à-pair* (P2P), où tous les calculateurs jouent le même rôle, sans avoir recours à une quelconque centralisation. Le système P2P résultant se nomme `SatCiety` [18].

Enfin, une étude sur le partitionnement statique de formules CNF a été proposée [11]. Ce partitionnement, dans lequel la formule CNF est représentée sous forme d'arbre, est donc effectué avant le début de la recherche. Ce type de partitionnement statique n'est pas connu pour être la technique la plus efficace pour l'obtention de formules équilibrées (i.e. de difficulté équivalente), mais les auteurs défendent cette façon de faire en arguant qu'une fois découpée et générée, une sous-formule (de type CNF) peut être fournie à n'importe quel solveur sans aucune modification de son code, le considérant ainsi comme une boîte noire.

Dans la section suivante, notre nouvelle plate-forme distribuée, baptisée `Dolius`, est présentée.

4 Détails techniques de `Dolius`

Dans cette section, nous détaillons les différents mécanismes et structures de données implémentés dans `Dolius`.

Le but de cette implémentation est de fournir une plateforme capable de résoudre une instance du problème SAT en distribué. Pour cela, cette plateforme est destinée à diviser le travail par l'approche diviser pour régner. De plus, il est également possible de transmettre une partie des connaissances accumulées dans l'espoir de limiter au maximum le travail redondant.

4.1 Structure générale / Initialisation

La structure générale de `Dolius` est de type maître/esclave. Les esclaves sont des solveurs de type CDCL ; dans la pratique nous utilisons `PeneLoPe` [1], ce qui nous permet d'obtenir un solveur distribué dont les esclaves peuvent être des solveurs séquentiels "classiques", ou des portofolios de solveurs, permettant d'exploiter au mieux les architectures multi-cœurs [1] ; tandis que le maître est un processus dont la tâche est de mettre en relation les esclaves, quand l'un d'eux a terminé sa tâche (voir plus bas).

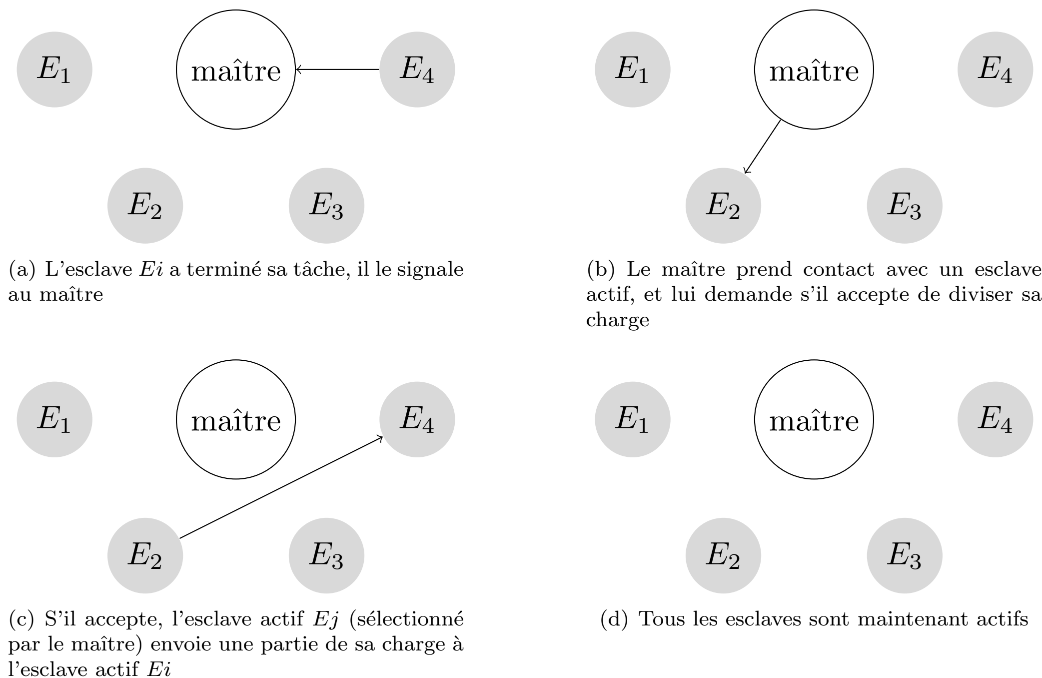
L'initiation de `Dolius` consiste simplement à démarrer le maître. Celui-ci est alors en attente d'un ou plusieurs esclaves, qui entameront alors la résolution du problème. Il est à noter que l'implémentation `Dolius` autorise l'ajout de ressources (esclaves) en cours de résolution. Ainsi, il est possible à n'importe quel moment de la recherche d'augmenter le nombre d'esclaves, ceux-ci n'ayant qu'à contacter le maître. Au travers de notre stratégie, une formule est prouvée satisfiable (SAT) si l'un des esclaves trouve un modèle. En effet, chaque esclave travaille avec la formule initiale simplifiée par la division du travail (voir section 4.3). Une formule est prouvée insatisfaisable (UNSAT) si tous les esclaves ont prouvé que la sous-formule dont ils s'occupent est UNSAT.

4.2 Rééquilibrage de charge

Il semble impossible en pratique de partitionner l'espace de recherche de manière à la fois statique et optimale au début de l'algorithme. Idéalement, les esclaves doivent avoir une charge de travail équivalente, mais il est en effet très difficile de prédire à l'avance la difficulté d'une (sous-)formule.

Un mécanisme de rééquilibrage de charge de travail doit donc être mis en place, afin d'être opportuniste vis-à-vis des ressources disponibles. En effet, certains esclaves termine en pratique leur tâche bien avant certains autres, dans la large majorité des cas. Nous avons ainsi implanté au sein de notre plate-forme un mécanisme qui permet de répartir les tâches sur les différentes unités de calcul, ceci à l'aide d'un processus de

1. Nous travaillons sur une API qui permettra de distribuer `Dolius` afin que chacun puisse y greffer son propre solveur SAT comme esclave



(a) L'esclave Ei a terminé sa tâche, il le signale au maître

(b) Le maître prend contact avec un esclave actif, et lui demande s'il accepte de diviser sa charge

(c) S'il accepte, l'esclave actif Ej (sélectionné par le maître) envoie une partie de sa charge à l'esclave actif Ei

(d) Tous les esclaves sont maintenant actifs

FIGURE 1 – Illustration de l'équilibrage de charge via le vol de travail

plus en plus répandu : le *vol de travail*. L'avantage de cette technique est qu'un esclave cherchant une solution ne doit pas se soucier d'autre esclaves qui seraient affamés. Ceux-ci viendront d'eux même demander du travail. Les différentes étapes de ce processus sont illustrées en Figure 1. Lorsqu'un esclave (noté Ei) termine la tâche qui lui est confiée, il contacte le maître (M) pour signaler qu'il n'a pas plus de travail (étape 1(a)). Le maître demande à l'un des esclaves encore actifs (noté Ej) s'il accepte de diviser sa tâche (étape 1(b)). Si Ej accepte, il contacte alors Ei pour se délester d'une partie de sa charge de travail (étape 1(c)). Les esclaves entrent donc directement en communication l'un avec l'autre sans passer par le maître, qui n'est contacté que pour fournir à un esclave nouvellement inactif les coordonnées d'un autre esclave acceptant de diviser sa tâche.

Dans notre plate-forme, un esclave ne peut refuser de diviser son travail que lorsqu'un des trois cas suivants se présentent :

1. l'esclave vient de trouver un modèle à la formule, la recherche d'une solution devient donc inutile
2. il n'œuvre sur sa tâche que depuis peu de temps (en pratique $< x$ secondes, où x peut être spécifié par l'utilisateur. Par défaut, $x = 2,5$)
3. il est déjà en cours de division de sa charge avec un autre esclave inactif

En outre, au sein de `Dolius`, le maître conserve l'ensemble des esclaves actifs dans une file. Lorsqu'un esclave demande du travail, il sélectionne le premier esclave actif de sa file afin de le contacter. Ce choix a été fait pour éviter, en cas de demandes de travail simultanées par plusieurs esclaves inactifs, qu'un même esclave soit contacté pour plusieurs demandes de division de travail, ce qui serait très inefficace.

Dans la section suivante, nous détaillons les stratégies de division du travail offertes par `Dolius`.

4.3 Division du travail

Lorsqu'il en reçoit la requête, la manière dont un esclave divise sa charge de travail pour en céder une partie est un facteur important de l'efficacité d'un algorithme de type diviser pour régner. La division sera considérée comme parfaite si les temps de résolution des sous-formules sont égaux entre eux, et inférieur au temps nécessaire pour résoudre la formule complète φ. Si la division est mal effectuée, deux problèmes de nature différente peuvent survenir. Premièrement, il est possible que les sous-formules φ_i aient un même temps de résolution que la formule complète. De ce fait, le temps nécessaire pour résoudre l'instance augmentera en fonction du nombre de ressources : le temps nécessaire pour résoudre l'instance auquel il faut ajouter le temps nécessaire pour la division du travail. Le deuxième problème lié à une mauvaise division se produit si le temps de résolution de φ_i est négligeable ou bien moindre que celui pour φ_j. Cela implique pour le système d'être capable de mettre en place une po-

litique de ré-équilibrage de travail. Si cette politique souffre de mauvaise division, des cas pathologiques peuvent apparaitre, comme le cas dit *ping-pong* [13].

Exemple 1 *Soit ϕ une formule CNF. La formule $\Sigma = ((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ est également une CNF. Les divisions sur les variables a et b posent toutes les deux problème.*

(a) si la division du travail de Σ est réalisée sur la variable a, l'une des deux charges de calculs est alors très faible, car il est très facile de prouver que $\Sigma \models a$. Ainsi, par simple propagation unitaire, il est possible de montrer que $\Sigma \wedge (\neg a) \models \bot$. L'esclave ayant reçu cette partie du problème va donc la prouver incohérente sans la moindre exploration (la propagation unitaire étant suffisante), et redemander du travail très rapidement. Ceci est problèmatique, car la division du travail a un coût, notamment en transfert réseau. Le phénomène ping-pong est le fait de multiplier de tels mauvais choix pour la division du travail, menant à des sous-problèmes d'une trivialité telle que certains esclaves passent plus de temps à demander du travail qu'à véritablement participer à la résolution de la formule.

(b) si la division est réalisée sur b, alors chacun des deux esclaves travaille sur la même formule : $a \wedge \phi$. En outre, si ϕ est UNSAT, c'est seulement lorsque le moins efficace des deux (i.e. celui répondant en dernier) transmet sa solution que l'incohérence de Σ peut être établie.

Ainsi, dans une telle situation, il est largement préférable de diviser selon l'une des variables de ϕ plutôt que sur a ou b.

Dans l'idéal, la tâche doit être divisée en deux sous-tâches différentes de difficulté similaire, afin de répartir au mieux la charge de travail, tout en s'assurant que les sous-tâches soient de difficulté moindre que la tâche avant division. Malheureusement, comme précisé plus haut, il semble difficile sinon impossible de déterminer à l'avance la difficulté d'une formule. Les algorithmes diviser pour régner basent donc leur division sur des concepts heuristiques.

Les algorithmes de ce type utilisent généralement la notion de *guiding path* pour la division du travail. La plupart du temps le *guiding path* est réduit à la simple *séparation* d'une variable, et affecte (pour lui même) l'une des variables de la formule tandis qu'il transmet à l'esclave inactif l'opposé de cette même variable [15, 12].

Notre plate-forme permet ce genre de division, mais se veut plus générique. En effet, `Dolius` permet de diviser sa charge de travail selon une formule booléenne ϕ quelconque, l'un des esclaves considérant cette formule ϕ vraie, tandis que l'autre fait l'hypothèse que sa négation $\neg\phi$ est vraie. La division du travail a donc pour effet d'obtenir un arbre de résolution dont la racine est la formule initiale Σ et les feuilles les formules simplifiées par les *guiding paths* ϕ_i successifs. Un exemple d'un tel arbre est montré dans la Figure 2

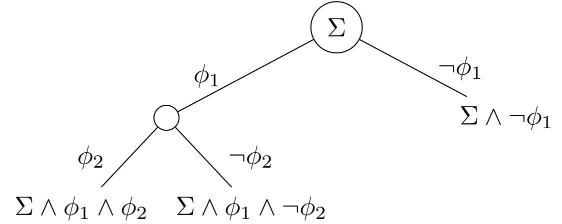


FIGURE 2 – Une exemple de division de travail

Formellement, l'ensemble des *guiding paths* doit vérifier les conditions suivantes :

Propriété 1 *Soit Σ la formule propositionnelle à résoudre et $\phi_1, \phi_2, \ldots \phi_n$ les guidings paths successifs utilisés pour diviser Σ. La condition suivante doit alors être vérifiée : $\Sigma \wedge \bigwedge_{i=1}^{n} \phi_i$ est SAT.*

Triviale dans le cas d'une division simple, cette propriété ne l'est plus dans le cas de divisions en utilisant des formules plus complexes. Nous étudions en ce moment des *guiding paths* dont la structure même nous assurent qu'ils sont mutuellement consistants. Clairement, qu'il s'agisse d'une simple variable ou d'une formule booléenne, le choix du *guiding path* est un élément déterminant pour l'efficacité de la procédure en général. Dans la première version de notre plate forme, nous avons fait le choix de conserver une division simple. Ainsi, l'esclave recevant une requête de division de travail sélectionne la variable ayant le VSIDS le plus important [15]. Cette valeur heuristique, utilisée au sein des solveurs CDCL comme choix de variable, est connue pour désigner les variables centrales de la formule [19]. Ainsi, diviser sur la variable présentant le score le plus important a pour but de scinder la formule sur une variable jugée pertinente.

En pratique, chaque esclave est une instance de `Penelope` travaillant sur plusieurs cœurs. Une difficulté supplémentaire apparait donc quand au choix de la variable de séparation. Pour l'instant, nous sélectionnons le cœur ayant généré le plus de conflits et sélectionnons sa "*meilleure*" variable. Il est clair que ce choix heuristique doit être amélioré et des expérimentations sont actuellement réalisées afin de déterminer un meilleur critère de sélection. Mais, à terme, nous souhaitons vivement déterminer une stratégie de séparation utilisant des formules plus complexes qu'une simple clause unitaire.

Instance	SAT ?	Penelope	5 esclaves			10 esclaves			20 esclaves		
			min	moy	F	min	moy	F	min	moy	F
ACG-15-5p1	O	97	101	127	0	93	118	2	79	112	1
traffic_3_uc_sat	O	43	90	209	0	82	105	0	77	85	0
q_query_3_L70_coli.sat	O	74	81	2485	1	63	88	1	80	84	0
q_query_3_L150_coli.sat	N	150	288	368	0	319	440	0	345	534	0
traffic_pcb_unknown	N	269	239	301	0	194	279	0	176	221	0
SAT_dat.k80	N	770	-	-	5	-	-	5	-	-	5
sortnet-8-ipc5-h19-sat	O	752	1173	1173	4	720	720	4	-	-	5
aes_64_1_keyfind_1	O	-	568	568	4	128	372	1	716	718	3
partial-10-13-s	O	-	576	797	3	185	284	3	376	470	2
AProVE07-01	N	-	-	-	5	1101	1164	2	558	638	1
UTI-20-10p1	O	-	-	-	5	-	-	5	766	766	4
eq.atree.braun.12.unsat	N	-	-	-	5	-	-	5	354	354	4

Table 1 – Résumé des résultats pour 12 instances. Pour chaque instance on affiche si elle est satisfiable, le temps de résolution nécessaire à `Penelope`, et le temps minimum (min) moyen sur les runs réussis (moy) et le nombre d'echecs (F) pour 5, 10 et 20 esclaves.

4.4 Transfert d'informations

Il est connu que la gestion des clauses apprises est un composant très important des solveurs CDCL [3] et des approches coopératives comme `ManySAT` ou `Penelope` [10, 1]. Dans le cas de notre approche distribuée ce problème perdure. Comme nous le montrerons dans la partie expérimentale, les performances de `Dolius` s'écroulent si l'esclave en manque de travail ne reçoit pas de clauses apprises en plus du *guiding path*. En effet, l'esclave nouvellement créé doit alors recommencer la recherche depuis le début et ne profite d'aucun effort fait précédemment. Dès lors, il faut déterminer quelles clauses apprises les esclaves partagent. Dans cette première version de `Dolius` + `PeneLoPe`, nous avons décider de partager toutes les clauses ayant un LBD [3] inférieur à 15. Les critères de sélection des no-goods à partager peuvent influer grandement, et des études plus approfondies sont encore nécessaires pour optimiser ce composant de `Dolius`.

Un autre type d'information que nous n'avons malheureusement pas eu le temps de mettre en œuvre concerne l'initialisation de l'heuristique : il semble en effet intéressant d'initialiser la recherche dans le même état que l'esclave qui partage son travail. La structure de `Dolius` permet de telles approches, que nous prévoyons de tester dans de futurs travaux. C'est une des nombreuses perspectives offertes par `Dolius`.

5 Évaluation expérimentale

Les expérimentations conduites dans cet article sont réalisées sur des bi-processeurs `Intel XEON X5550` 4 cœurs à 2.66 GHz avec 8Mo de cache et 32 Go de RAM, sous Linux CentOS 6 (kernel 2.6.32). Chaque esclave utilise 8 *threads*. Le temps limite alloué pour résoudre une instance est de 1200 secondes WC (Wall Clock) (Les temps seront toujours donnés en secondes). Nous considérons ici le temps réel (WC) plutôt que le temps CPU car ce dernier n'intègre que le temps d'activité des unités de calcul, et occulte donc certains aspects, comme les temps d'attente des processus lors des communications réseau. Nous avons choisi un pool de 12 instances de difficulté variable issues de la compétition SAT 2011. Chaque instance a été testée avec 5, 10 et 20 esclaves.

Précisons qu'il est délicat d'évaluer expérimentalement une plate-forme de cette nature. C'est probablement l'une des raisons pour lesquelles depuis des années, aucun solveur de type diviser pour régner n'est soumis aux compétitions SAT. En effet, ce type d'approche n'est pas déterministe par nature, ni en temps d'exécution, ni dans la solution (preuve de réfutation ou modèle) reportée. Plusieurs exécutions de `Dolius` + `PeneLoPe` sur un même ensemble de machines peuvent amener à des résultats disparates, dont la variance peut être plus ou moins grande. Ceci est inhérent à la plupart des processus distribués en informatique. Afin d'obtenir des résultats viables, nous avons donc lancé chaque instance à 5 reprises.

La table 1 donne une vue globale de l'ensemble des résultats obtenus. Nous avons également mis les résultats obtenus par `Penelope` (correspondant à `Dolius` avec un seul esclave).

Jetons tout d'abord un œil aux 5 premières instances de ce tableau. Elles sont (relativement) faciles pour `PeneLoPe` et il semble que l'approche distribuée ne soit pas adaptée à ce type de formules simples. Pour

					Attente		
E	Temps	Mo	req	dernière	somme	moyenne	médiane
5	-	-	56	19.77	66.70	13.34	13.70
5	-	-	46	14.18	50.75	10.15	11.02
5	-	-	46	14.30	51.13	10.23	11.00
5	576	2	47	15.32	63.90	12.78	12.79
5	1019	1	48	14.10	50.23	10.05	10.92
10	-	-	141	27.30	143.47	14.35	14.93
10	-	-	160	32.97	162.61	16.26	16.79
10	-	-	211	31.40	201.61	20.16	20.86
10	383	4	164	42.26	175.99	17.60	18.17
10	185	2	182	24.37	161.81	16.18	16.49
20	-	-	472	35.15	376.62	18.83	19.16
20	-	-	466	37.34	352.25	17.61	18.80
20	637	8	415	30.82	344.25	17.21	17.21
20	376	7	438	49.17	330.61	16.53	16.95
20	398	5	479	21.05	325.88	16.29	17.58

Table 2 – Détail des résultats pour l'instance partial-10-13-s. Chaque ligne correspond à un lancement. On y reporte le nombre d'esclaves (E), Le temps nécessaire à la résolution, le nombre de demandes de travail (req), la quantité totale transférée sur le réseau en mégaoctet (Mo), La dernière demande de travail par un esclave, la somme, moyenne, et médiane des temps d'attente de chaque esclave.

de tels problèmes, notre implémentation fournit même de moins bons résultats qu'une approche centralisée, et dans certains cas, l'approche distribuée est d'ailleurs incapable de répondre à chaque exécution. Les 2 instances qui suivent (SAT_dat.k80 et sortnet-8-ipc5-h19-sat) présentent plus de difficultés pour `PeneLoPe`. Ici, `Dolius` obtient des résultats décevants, puisque dans les deux cas, l'approche contenant 20 esclaves ne parvient jamais à résoudre l'instance. Les versions contenant moins d'esclaves réussissent quant à eux une seule fois.

Les dernières instances listées dans la table 1 sont en revanche des instances difficiles, pour lesquels `PeneLoPe` ne réussit pas à déterminer la cohérence en moins de 1200 secondes. On commence à voir ici l'intérêt d'une approche distribuée puisque sur de telles instances, l'utilisation d'un certain nombre d'esclaves (chacun d'eux étant basés sur `PeneLoPe`) permet de trouver une solution là où l'approche portofolio seule montre ses limites. En effet, l'utilisation de ces ressources supplémentaires permet l'obtention d'une réponse lors de certaines exécutions (d'autres terminent en échec), en des temps de calcul très raisonnables pour des problèmes de cette difficulté.

Encore une fois, `Dolius` est un projet de longue haleine, dont nous ne présentons ici qu'un travail préliminaire. Il est clair que `Dolius` peut être plus finement paramétré afin d'accroître son efficacité pratique, toutefois les résultats obtenus par sa première version sont très prometteurs.

La table 2 donne des détails sur l'ensemble des runs de l'instance partial-10-13-s. Malheureusement, lorsque les instances ne sont pas résolues en 1200 secondes (-) nous n'avons pas la possibilité d'afficher les transferts réseaux réalisés. Comparons tout d'abord les lancements avec des nombres d'esclaves différents. Il est clair que le nombre de demandes de travail et la quantité transférée sur le réseau augmente avec le nombre de travailleurs. Il est par contre difficile de mettre en relation le temps d'exécution avec le nombre de travailleurs : nous ne sommes actuellement pas en mesure de proposer une méthode assurant une diminution du temps de calcul en augmentant le nombre de travailleurs.

La table 2 donne par contre des indications très intéressantes sur la division du travail. En effet, pour un nombre de travailleurs donné, le nombre de requêtes demandées et le temps où tous les esclaves travaillent jusqu'à la fin sont relativement proches y compris lorsque le solveur échoue à trouver une solution après 1200 secondes. Dans ce cas, la division de travail échoue quelque peu : Aucun espace de recherche n'est prouvé insatisfiable et l'affectation de certaines variables n'aide pas à la détection d'un modèle ; d'où l'importance du choix du *guiding path.*

6 Conclusion

Dans cet article, nous avons proposé un cadre permettant de résoudre le problème SAT de manière pa-

rallèle. Cette première version est capable de résoudre des instances très difficiles pour les solveurs centralisés. Il est de toute façon clair qu'une approche distribuée n'est utile que pour des instances très difficiles, le surcoût dû aux différents demandes de travail, transfert, etc. s'avérant rédhibitoires sur des instances dont le temps de résolution séquentiel est relativement court.

Clairement, le choix du *guiding path* est un élément important pour un tel algorithme. La séparation sur une seule variable (celle avec le VSIDS le plus élevé) permet d'obtenir des premiers résultats relativement satisfaisants. Il est toutefois clair qu'il est possible d'améliorer l'efficacité de la procédure en divisant avec soin le travail à accomplir. L'utilisation de formules plus générales que de simples clauses unitaires constitue également un point qu'il est important d'étudier.

De manière plus générale, ce travail offre de nombreuses perspectives de travail. Le choix de l'information à transférer entre esclaves, le temps avant qu'un esclave accepte de partager son travail, l'étude des cas pathologiques comme le cas du *ping-pong* sont autant de pistes de recherches que nous allons étudier dans les prochains mois.

Enfin, en mettant en place une API qui permettra à tout un chacun d'utiliser la plateforme `Dolius`, nous espérons contribuer à l'étude des solveurs SAT sur les environnements distribués.

Références

[1] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Proceedings of SAT'12*, pages 200–213, 2012.

[2] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *proceedings of SAT'11*, pages 147–160, 2011.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of IJCAI'09*, pages 399–404, 2009.

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

[5] Wahid Chrabakh and Richard Wolski. The gridsat portal : a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation : Practice and Experience*, 19(6) :795–808, 2007.

[6] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of STOC'71*, 1971.

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

[8] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver : Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3) :75–90, 2005.

[9] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *Proceedings of IJCAI'09*, pages 499–504, 2009.

[10] Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Manysat : a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262, 2009.

[11] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of LPAR'10*, pages 372–386, 2010.

[12] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *Proceedings of CP*, pages 385–399, 2011.

[13] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9 :174–189, 2001.

[14] Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD'96*, pages 220–227, 1996.

[15] Ruben Martins, Vasco Manquinho, and Ines Lynce. Improving search space splitting for parallel SAT solving. In *Proceedings of ICTAI'10*, pages 336–343, 2010.

[16] Mikhail Posypkin, Alexander Semenov, and Oleg Zaikin. `SAT@HOME` web page. `http://sat.isa.ru/pdsat`, 2008.

[17] C.B. Ries. *BOINC : Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing.* Springer, 2012.

[18] Sven Schulz and Wolfgang Blochinger. Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing*, 8(3) :443–471, 2010.

[19] Laurent Simon and George Katsirelos. Eigenvector centrality in industrial SAT instances. In *Proceedings of CP'12*, pages 348–356, 2012.

[20] Lintao Zhang, Connor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD'01*, pages 279–285, 2001.