Une nouvelle architecture parallèle pour le problème de validité des QBF

Benoit da Mota, Pascal Nicolas, Igor Stéphan

▶ **To cite this version:**

Benoit da Mota, Pascal Nicolas, Igor Stéphan. Une nouvelle architecture parallèle pour le problème de validité des QBF. JFPC 2010 - Sixièmes Journées Francophones de Programmation par Contraintes, Jun 2010, Caen, France. pp.113-122. inria-00520314

HAL Id: inria-00520314
https://inria.hal.science/inria-00520314

Submitted on 22 Sep 2010

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

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

Une nouvelle architecture parallèle pour le problème de validité des QBF

Benoit Da Mota, Pascal Nicolas, Igor Stéphan

LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045, Angers, Cedex 01, France
{damota|pn|stephan}@info.univ-angers.fr

Résumé

Dans ce papier, nous présentons une nouvelle architecture parallèle ouverte pour répondre à différents problèmes portant sur les formules booléennes quantifiées, dont la validité. La principale caractéristique de notre approche est qu'elle est basée sur un découpage syntaxique de la formule pour le traitement de sous-formules indépendantes. Elle est liée au choix de traiter des formules booléennes quantifiées sans restriction syntaxique, comme la forme prénexe ou la forme normale conjonctive. Dans ce cadre parallèle général ouvert, nous sommes capables d'introduire différents oracles, sous la forme d'algorithmes séquentiels pour l'élimination de quantificateurs. Nous présentons nos premières expérimentations en instanciant un unique oracle et en rapportons les résultats.

1 Introduction

Le problème de validité pour les formules booléennes quantifiées (QBF) est une généralisation du problème de satisfiabilité pour les formules booléennes. Tandis que décider de la satisfiabilité des formules booléennes est NP-complet, décider de la validité des QBF est PSPACE-complet. C'est le prix à payer pour une représentation plus concise pour de très nombreuses classes de formules. Une multitude d'importants problèmes de décision parmi des champs très divers ont des transformations polynomiales vers le problème de validité des QBF : planification [22, 2], construction de modèles bornés [2], vérification formelle (voir [4] pour une vue d'ensemble).

La plupart des procédures récentes et efficaces pour décider de la validité des QBF, nécessitent d'avoir en entrée un formule sous forme normale négative ou dans une forme encore plus restrictive comme la forme normale conjonctive. Mais il est rare que les problèmes s'expriment directement sous ces formes qui détruisent complètement la structure originale des problèmes. Il est plus naturel d'utiliser toute l'expressivité du langage QBF : tous les connecteurs logiques usuels, y compris l'implication, la bi-implication et le ou-exclusif, ainsi que la possibilité d'utiliser des quantificateurs à l'intérieur de la formule. Aussi, notre premier objectif est de développer un solveur QBF sans restriction sur les formules en entrée.

Depuis quelques années, la fréquence du ou des cœurs des processeurs ne progresse plus, voire même diminue. En contrepartie, le nombre de cœurs par processeur augmente et le domaine de la programmation parallèle est en pleine effervescence : multithread, multi-CPU, GPU computing, HPC, grid computing, cloud computing, etc. Certaines procédures de résolution profitent déjà de ce nouveau potentiel et les futurs solveurs devront en faire de même pour rester compétitifs. Le second objectif de notre travail est d'exploiter les opportunités qu'offre la programmation parallèle pour s'attaquer au problème de validité des QBF en réutilisant, adaptant, améliorant ou combinant des algorithmes séquentiels pour QBF à l'intérieur d'une architecture parallèle.

Par ailleurs, au delà du problème de validité, nous nous intéressons aussi à d'autres problèmes en rapport avec QBF, tels celui de leur compilation pour une représentation plus compacte et un accès de complexité moindre aux solutions. Dans ce cas, une procédure de décision ne suffit plus (la réponse du système n'étant plus simplement « oui » ou « non »), mais un ensemble de formules décrivant l'espace des solutions pour les symboles propositionnels existentiellement quantifiés. Aussi dans ce but, notre objectif à long terme est de réaliser une architecture aussi ouverte que possible pour offrir à l'utilisateur final un ensemble d'outils en rapport avec les QBF.

Ce présent article est structuré ainsi : après un rap-

pel des notions fondamentales concernant la logique propositionnelle, le problème SAT et la définition de la syntaxe et de la sémantique des QBF, la section 3 dresse un court panorama des procédures de décision pour le problème de validité des QBF dans le cadre des solveurs séquentiels puis parallèles. La section 4 décrit notre nouvelle architecture parallèle pour le problème de validité des QBF comme une architecture maître/esclave dont le maître assure la répartition de tâches aux esclaves obtenues par un découpage syntaxique de la formule initiale. La section 5 propose une instance de notre nouvelle architecture selon un mode client/serveur : le client est une implantation de l'esclave basée sur la procédure de décision par oracle `QSAT` ; cette section détaille les choix techniques ainsi que les résultats préliminaires de notre approche. La section 6 dresse une conclusion et des perspectives à ce travail.

2 Préliminaires

2.1 Logique propositionnelle

L'ensemble des valeurs booléennes **vrai** et **faux** est noté **BOOL**. L'ensemble des symboles propositionnels est noté $\mathcal{SP}$. Les symboles $\top$ et $\bot$ sont les constantes booléennes. Le symbole $\wedge$ est utilisé pour la conjonction, $\vee$ pour la disjonction, $\neg$ pour la négation, $\rightarrow$ pour l'implication, $\leftrightarrow$ pour la bi-implication et $\oplus$ pour le ou-exclusif. L'ensemble des opérateurs binaires $\{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ est noté $\mathcal{O}$. L'ensemble **PROP** des formules propositionnelles est défini inductivement ainsi : tout symbole propositionnel ou constante propositionnelle est élément de **PROP** ; si F est élément de **PROP** alors $\neg F$ est élément de **PROP** ; si F et G sont éléments de **PROP** et $\circ$ est élément de $\mathcal{O}$ alors $(F \circ G)$ est élément de **PROP**. Un littéral est un symbole propositionnel ou la négation de celui-ci. Une clause est une disjonction de littéraux. Une formule propositionnelle est sous forme normale négative (FNN) si la formule n'est constituée exclusivement que de conjonctions, disjonctions et littéraux. Une formule propositionnelle est sous forme normale conjonctive (FNC) si c'est une conjonction de disjonctions de littéraux. Toute formule sous FNC est une formule sous FNN. Une valuation v est une fonction de $\mathcal{SP}$ dans **BOOL** (l'ensemble des valuation est noté **VAL**).

2.2 Syntaxe des formules booléennes quantifiées

Le symbole $\exists$ est utilisé pour la quantification existentielle et $\forall$ pour la quantification universelle (q est utilisé pour noter un quantificateur quelconque). L'ensemble **QBF** des formules booléennes quantifiées est défini inductivement ainsi : si F est un élément de **PROP** alors c'est un élément de **QBF** ; si F est un élément de **QBF** et x est un symbole propositionnel alors $(\exists x\ F)$ et $(\forall x\ F)$ sont des éléments de **QBF** ; si F est élément de **QBF** alors $\neg F$ est élément de **QBF** ; si F et G sont éléments de **QBF** et $\circ$ est élément de $\mathcal{O}$ alors $(F \circ G)$ est élément de **QBF**. Un symbole propositionnel x est libre s'il n'apparaît pas sous la portée d'un quantificateur $\exists x$ ou $\forall x$. Une QBF est close si elle ne contient pas de symbole propositionnel libre. Une substitution est une fonction de l'ensemble des symboles propositionnels dans l'ensemble des formules (quantifiées ou non). Nous définissons la substitution de x par F dans G, notée $[x \leftarrow F](G)$, comme étant la formule obtenue de G en remplaçant toutes les occurrences du symbole propositionnel x par la formule F sauf pour les occurrences de x sous la portée d'un quantificateur portant sur x. Un lieur est une chaîne de caractères $q_1 x_1 \dots q_n x_n$ avec $x_1, \dots, x_n$ des symboles propositionnels distincts et $q_1, \dots, q_n$ des quantificateurs. Une QBF QM est sous forme prénexe si Q est un lieur et M est une formule booléenne, appelée matrice. Une QBF prénexe QM est sous forme normale négative (resp. conjonctive) si M est une formule booléenne en FNN (resp. FNC).

2.3 Sémantique des formules booléennes quantifiées

La sémantique des QBF présentée fait appel à la sémantique des (constantes et) opérateurs booléens qui est définie de manière habituelle, en particulier, à chaque (constante et) opérateur (resp. $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\oplus$) est associée une fonction booléenne (resp. $i_\top, i_\bot :\rightarrow$ **BOOL**, $i_\neg$: **BOOL** $\rightarrow$ **BOOL**, $i_\wedge, i_\vee, i_\rightarrow, i_\leftrightarrow, i_\oplus$: **BOOL** $\times$ **BOOL** $\rightarrow$ **BOOL**) qui en définit sa sémantique ; la sémantique des QBF fait aussi appel à une sémantique pour les quantificateurs :

$$I^x_\exists, I^x_\forall : (\mathbf{VAL} \rightarrow \mathbf{BOOL}) \times \mathbf{VAL} \rightarrow \mathbf{BOOL}$$

$$I^x_\exists(f)(v) = i_\vee(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

$$I^x_\forall(f)(v) = i_\wedge(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

enfin elle est définie inductivement :

- $[[\bot]](v) = i_\bot$;
- $[[\top]](v) = i_\top$;
- $[[x]](v) = v(x)$ si $x \in \mathcal{SP}$;
- $[[(F \circ G)]](v) = i_\circ([[F]](v), [[G]](v))$ si $F, G \in$ **QBF** et $\circ \in \mathcal{O}$;
- $[[\neg F]](v) = i_\neg([[F]](v))$ si $F \in$ **QBF** ;
- $[[(\exists x\ F)]](v) = I^x_\exists([[F]])(v)$ si $F \in$ **QBF** ;
- $[[(\forall x\ F)]](v) = I^x_\forall([[F]])(v)$ si $F \in$ **QBF**.

Une QBF close F est valide si $[[F]](v) =$ **vrai** pour toute valuation v. Par exemple la QBF

$\exists a \exists b \forall c((a \vee b) \leftrightarrow c)$ n'est pas valide tandis que la QBF $\forall c \exists a \exists b((a \vee b) \leftrightarrow c)$ l'est. Cet exemple montre que l'ordre des quantificateurs est crucial pour décider de la validité d'une QBF.

Comme dans le cas propositionnel, une relation d'équivalence notée $\equiv$ est définie pour les QBF par $F \equiv G$ si $[[F]](v) = [[G]](v)$ pour toute valuation v. En lien avec l'exemple qui précède, $\exists x \exists y F \equiv \exists y \exists x F$ et $\forall x \forall y F \equiv \forall y \forall x F$ mais $\exists a \exists b \forall c((a \vee b) \leftrightarrow c) \not\equiv \forall c \exists a \exists b((a \vee b) \leftrightarrow c)$.

Enfin, rappelons que le problème (SAT) consistant à décider si une formule booléenne est satisfiable ou non est le problème canonique de la classe NP-complet. De son côté, le problème consistant à décider si une formule booléenne quantifiée est valide ou non est le problème canonique de la classe PSPACE-complet [28].

3 État de l'art des solveurs sequentiels/parallèles pour QBF

État de l'art des procédures de décision pour le problème de validité des QBF. Puisque la vérification d'un modèle pour une QBF prénexe est co-NP-complet [16], il y a très peu de procédures incomplètes basées sur des métaheuristiques. Pour autant que nous sachions, il y en a deux qui sont basées sur la recherche locale : `WalkQSAT` [12] (basé sur `WalkSAT` [25]) et `QBDD(LS)` [1]. Ainsi, la plupart des procédures pour le problème de validité des QBF sont des procédures de décision et elles peuvent être partitionnées en trois catégories : les procédures dites « monolithiques » qui se suffisent à elles-mêmes, les procédures qui transforment le problème dans un autre formalisme possédant une procédure de décision et les procédures qui exploitent un oracle.

Dans la première catégorie, les procédures sont basées sur la résolution telles que `QKN` [15], sont des procédures par élimination de symboles propositionnels à la Fourier-Motzkin telles que `quantor` [5] (pour des QBF FNC) ou `Nenofex` [19] (pour des QBF FNN) comme extension de l'algorithme de Davis et Putnam [8], ou sont des procédures de recherche par élimination des quantificateurs les plus externes telles que `Evaluate` [6], `decide` [23], `QUBE` [14] ou `QSOLVE` [10] (toutes pour des QBF FNC) ou `qpro` [9] (pour des QBF FNN) comme extension de l'algorithme de Davis, Logemann et Loveland [7].

Une procédure basée sur une transformation traduit la QBF dans un formalisme qui dispose déjà d'une procédure de décision efficace : SAT (en FNC) pour `sKizzo` [3] ou ASP [27] (dans ces deux cas avec une croissance potentiellement exponentielle de la formule).

Les procédures avec oracle reviennent à l'idée initiale de la « hiérarchie polynomiale » [20] en ce qu'un oracle capable de résoudre un sous-problème avec une complexité moindre est nécessaire : `QBDD(DLL)` [1] utilise un oracle NP-complet et `QSAT` [21] utilise deux oracles dont l'un est NP-complet et l'autre co-NP-complet. Cette dernière procédure est détaillée au paragraphe 5 car l'implantation de notre modèle parallèle s'instancie grâce à `QSAT`.

État de l'art pour les solveurs parallèles. Autant que nous le sachions, il existe trois implantations de solveurs parallèles pour le problème de validité des QBF : `PQSOLVE` [10], `PaQube` [17] et `QMiraXT` [18]. Nous faisons deux remarques importantes :

- Ces procédures sont toutes dédiées aux QBF sous FNC prénexes.
- Ces procédures sont toutes basées sur un algorithme de recherche séquentiel par élimination du quantificateur le plus externe vers le plus interne (`QSOLVE` [10] pour `PQSOLVE`, `QUBE` [14] pour `PaQube` et `PaMiraXT` [24], un solveur SAT parallèle, pour `QMiraXT`) et ainsi appliquent une *stratégie de partitionnement sémantique* qui choisit un symbole propositionnel du bloc de quantificateurs le plus externe et applique la sémantique des quantificateurs pour partitionner le problème et distribuer les tâches : si la QBF dont on désire décider de la validité est $qxQM$ alors les deux tâches $Q[x \leftarrow \top](M)$ et $Q[x \leftarrow \bot](M)$ sont distribuées.

La procédure `PQSOLVE` est un solveur distribué qui utilise les techniques de la parallélisation des programmes d'échec [10]. Il instancie un modèle pair-à-pair : un processus inactif demande du travail à un processus choisi au hasard et en devient l'esclave pour une tâche. Chaque processus a une pile de tâches à réaliser qui est augmentée par le partitionnement sémantique de celles qui sont estimées trop complexes ; le maître envoie une de ces tâches à son esclave momentané qui a sollicité une tâche à réaliser. Un esclave peut devenir lui-même le maître d'un autre processus.

La procédure `QMiraXT` est dédiée à la prise en compte du potentiel de performance des architectures modernes multi-cœur et/ou processeurs multithreadés. En utilisant un solveur threadé à mémoire partagée, les clauses apprises par conflit [29] sont partagées en les différents espaces de recherche. Il n'y a pas de processus maître mais à la place un « Master Control Object » (MCO) qui permet aux threads de communiquer via des messages asynchrones pour des évènements globaux (par exemple, si un sous-problème est valide ou non). Le MCO prend en charge aussi la stratégie de partitionnement sémantique, appelée dans ce cadre « Single Quantification Level Scheduling » (ou SQLS)

et distribue les tâches.

La procédure `PaQube` est conçue selon le modèle maître/esclave où un processus est dédié au maître (ce qui ne nécessite pas de CPU dédiée) et les autres aux esclaves qui réalisent en fait la recherche. `PaQube` est un solveur QBF parallèle basé sur MPI. Au travers de messages, les esclaves partagent certaines des clauses et cubes issus respectivement des conflits et solutions appris. Ainsi, chaque esclave cumule localement toute l'expertise obtenue par l'ensemble des autres esclaves. Enfin, le travail principal du maître est de réaliser, comme pour `QMiraXT`, la SQLS.

De par leur modèle, `PQSOLVE` et `PaQube` sont plus extensibles aux clusters et grids que `QMiraXT` mais ce dernier solveur semble tenir plus compte de l'évolution du matériel que les deux premiers.

4 La parallélisation syntaxique

Les solveurs parallèles présentés dans l'état de l'art de la section précédente sont tous basés sur un partitionnement sémantique des tâches distribuées selon une « architecture de parallélisation sémantique ». Nous proposons une nouvelle approche : l'« architecture de parallélisation syntaxique ». Celle-ci n'est pas basée sur le partitionnement de l'espace de recherche selon la sémantique des quantificateurs mais selon la localité de l'information dans les QBF non prénexes (et non FNC) représentée par des sous-formules quantifiées à symboles propositionnels libres.

4.1 L'extraction syntaxique de sous-problèmes

L'extraction syntaxique de sous-problèmes consiste à chercher des sous-formules que l'on peut traiter de manière quasiment indépendante, c-à-d. partageant peu de symboles propositionnels libres avec le reste de la QBF. Une sous-formule extraite est réduite par élimination des quantificateurs grâce à un oracle qui peut prendre des formes diverses selon les instanciations de l'architecture (comme nous le verrons à la section suivante où la procédure de décision `QSAT` est choisie) et qui en calcule une formule équivalente.

Par exemple, pour la QBF

$$(\exists a\ (a{\rightarrow}(\forall b\ (b{\rightarrow}((\forall c\ (c{\vee}b)){\wedge}(\exists d\ (a{\wedge}{\neg}d)))))))$$

l'extraction syntaxique calcule l'ensemble $\{(\forall c\ \ (c{\vee}b)), (\exists d\ \ (a{\wedge}{\neg}d))\}$ comme étant les sous-problèmes qui peuvent être traités séparément. Nous appelons *largeur syntaxique* la taille de l'ensemble obtenu par cette extraction syntaxique (celle-ci ne variant pas d'une exécution à l'autre, ceci permet de limiter les hypothèses lors de l'étude des résultats expérimentaux). Les oracles invoqués rendent les formules $b \equiv (\forall c\ (c{\vee}b))$ et $a \equiv (\exists d\ (a{\wedge}{\neg}d))$; ces formules sont réintroduites dans la QBF originale pour obtenir $(\exists a\ \ (a{\rightarrow}(\forall b\ \ (b{\rightarrow}(b{\wedge}a)))))$; à nouveau l'extraction syntaxique calcule un ensemble $\{(\forall b\ \ (b{\rightarrow}(b{\wedge}a)))\}$; l'oracle invoqué rend la formule $a \equiv (\forall b\ (b{\rightarrow}(b{\wedge}a)))$; finalement cette formule est réinjectée dans la formule intermédiaire pour obtenir la QBF $(\exists a\ (a{\rightarrow}a))$ qui est alors démontrée comme étant valide car équivalent à $\top$.

4.2 Un modèle maître/esclave

Une manière d'instancier l'architecture de parallèlisation syntaxique est de choisir un modèle maître/esclave. Ainsi le maître, dont le fonctionnement est illustré dans la figure 1 qui décrit la boucle générale d'exécution, est chargé d'assurer la répartition de tâches et la complétude de la recherche. Il lit la formule originale, extrait syntaxiquement des sous-problèmes, les distribue aux esclaves, puis attend les réponses et réinsère les résultats dans la formule d'origine. Pour le problème de validité des QBF, il suffit de répéter l'opération jusqu'à obtenir $\top$ ou $\bot$. Le rôle d'un esclave est alors d'accepter une QBF prénexe non FNC avec des symboles propositionnels libres et de répondre par un formule booléenne non quantifiée équivalente, uniquement composée des symboles propositionnels libres.

A chaque cycle d'extraction de sous-formules nous pouvons attribuer une *largeur syntaxique*. Nous appelons *largeur syntaxique maximale*, la plus grande valeur de largeur atteinte. Ce nombre traduit le besoin maximal en esclaves pour une formule et un découpage syntaxique donné. De même, nous appelons *largeur syntaxique minimale*, la plus petite valeur de largeur rencontrée. Selon le découpage syntaxique choisi, l'approche parallèle pourrait très vite être limitée. Tout esclave au delà de la largeur syntaxique maximale serait inutile. Or, de nombreux problèmes possèdent une largeur maximale de 1 ou 2 avec le découpage présenté dans la section 4.1.

Avant d'appliquer une méthode de résolution, le client utilise un heuristique de découpage (client splitting heuristic). Il a la responsabilité de créer des sous-problèmes si le problème qu'il reçoit lui parait trop difficile. Le client peut par exemple effectuer un découpage sémantique sur les symboles propositionnels libres et renvoyer un ensemble de tâches au maître. Si tous les clients sont occupés et que le problème est difficile, le client rajoute des tâches en file d'attente, si le problème semble facile, il le résout a priori rapidement et attend une nouvelle tâche. Si des esclaves sont en attente de travail et que le problème est difficile, le nouveau découpage va permettre au maître de

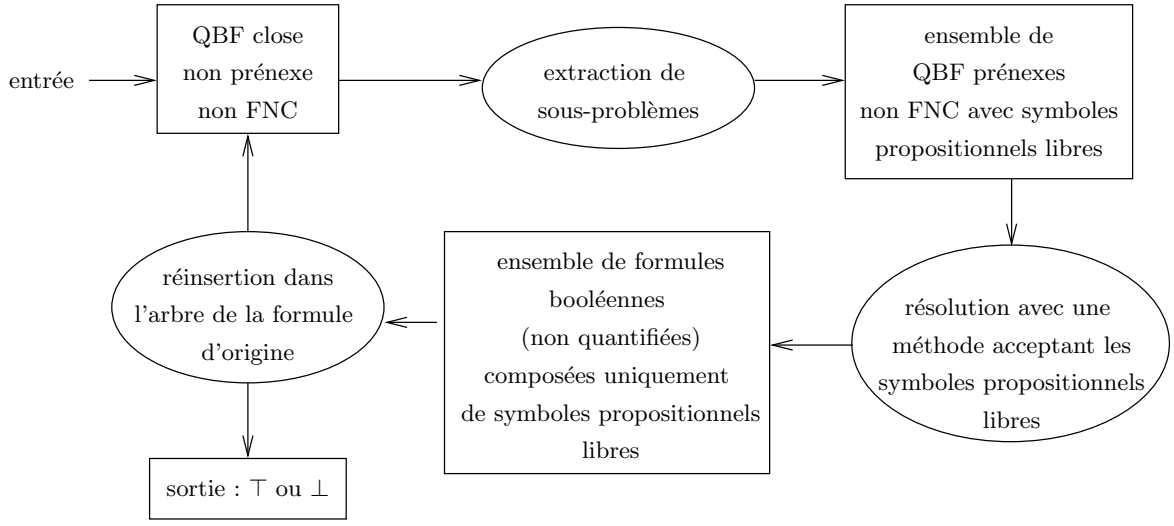

FIG. 1 – Boucle générale d'exécution.

redonner des tâches.

5 Une première instance de l'architecture de parallélisation syntaxique

Dans l'architecture présentée dans cet article, la tâche des esclaves est de calculer pour une QBF, une formule propositionnelle équivalente au sens de la préservation des modèles ; la QBF étant une sous-formule d'une QBF plus large, celle là est remplacée dans cette dernière par la proposition générée. Nous avons choisi pour instancier notre architecture parallèle un principe client/serveur sur un cluster de nœuds de calcul[1] et une procédure de l'état de l'art qui applique un partitionnement syntaxique de l'espace de recherche avec un oracle : la procédure `QSAT` [21]. Nous décrivons tout d'abord la procédure `QSAT` puis les résultats expérimentaux obtenus.

5.1 La procédure `QSAT`

`QSAT` [21] est une procédure de décision pour les QBF par élimination de quantificateurs, des plus internes vers les plus externes. Cette procédure opère itérativement sur une formule $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans F ; l'équivalence $(\exists x\ (F \wedge G)) \equiv (F \wedge (\exists x\ G))$ permet d'isoler la sous formule $(\exists x\ G)$ qui va être mise en équivalence logique par une procédure *simp* avec une formule G' ne contenant pas le symbole propositionnel x ; par substitution des équivalents $Q(\exists x\ (F \wedge G)) \equiv Q(F \wedge G')$. Le procédé est similaire avec une quantification universelle. Le processus est itéré jusqu'à élimination de tous les quantificateurs. La procédure *simp* a besoin d'une procédure de décision pour le problème SAT et d'une procédure de décision pour le problème TAUT pour être effective. Cette procédure construit une FNC G' sur les symboles propositionnels libres de G telle que $G' \equiv (\exists x\ G)$. Elle opère comme un Davis-Logemann-Loveland par séparation de l'espace de recherche sur un symbole propositionnel libre y de G par appel récursif sur $[y \leftarrow \top](G)$ et $[y \leftarrow \bot](G)$. Si $[y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G)$ est insatisfiable alors $simp((\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G)))$ retourne la clause $((y_1 \oplus C_1) \vee \ldots \vee (y_n \oplus C_n))$; sinon si $[y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G)$ est une tautologie alors $simp((\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon si $(\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G))$ ne contient plus de symbole propositionnel libre alors soit cette formule est insatisfiable (et ce cas a déjà été traité) soit elle est satisfiable et $simp((\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon un nouveau symbole propositionnel libre y_{n+1} est considéré et $G_\top = simp((\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n][y_{n+1} \leftarrow \top](G)))$ et $G_\bot = simp((\exists x\ [y_1 \leftarrow C_1] \ldots [y_n \leftarrow C_n][y_{n+1} \leftarrow \bot](G)))$ sont calculés, $(G_\top \wedge G_\bot)$ est retourné. L'algorithme de la procédure *simp* est basé sur la construction classique de la FNC comme étant la conjonction de la négation des lignes de la table de vérité (considérées comme des cubes) qui falsifient la négation de la formule. Seule l'implantation d'une double restriction de `QSAT` est décrite dans [21] : restrictions à SAT et aux formules sous FNC. Dans le cadre de cette dernière restriction, la recherche d'une forme $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans

[1] Ainsi, par la suite, nous utiliserons de manière indifférenciée les termes « serveur » ou « maître » et « client » ou « esclave ».

F est triviale ; seul le choix de x demeure et permet l'édification de stratégies.

5.2 Description du client

D'un point de vue général, un client reçoit une QBF quelconque avec des symboles propositionnels libres et doit renvoyer une formule booléenne non quantifiée composée uniquement de ces symboles propositionnels libres. La procédure QSAT, décrite dans le paragraphe précédent, s'y prête bien. De plus, il est possible d'éliminer plusieurs quantificateurs consécutifs du même type. Pour une QBF prénexe, il faudra autant d'itérations de la procédure QSAT que d'alternances de quantificateurs. La procédure QSAT possède un inconvénient majeur, elle est efficace sur une certaine catégorie de formules dites longues et fines (long and thin [21]). C'est pourquoi, d'une manière générale, notre architecture prévoit qu'un client puisse renoncer à résoudre une tâche qu'il juge trop difficile. La figure 2 décrit le fonctionnement d'un client.

Tout d'abord le client démarre et se met en attente d'une tâche. Puis il reçoit `MSG_JOB` suivi d'une tâche : une formule accompagnée d'une affectation partielle ou `MSG_JOB_SAME` suivi uniquement d'une affectation partielle. Le client appelle une fonction heuristique tentant d'évaluer rapidement la difficulté de la tâche reçue :

- La tâche est considérée comme abordable : la tâche est exécutée et un message est retourné :
  - `MSG_CONST` avec $\top$ ou $\bot$,
  - ou `MSG_OP` avec une formule propositionnelle seulement constituée des symboles propositionnels libres,
  - ou `MSG_CNF` avec une formule comme pour `MSG_OP` mais en FNC.
- La tache est considérée trop difficile a résoudre : alors un découpage sémantique est appliqué à la formule et le message `MSG_SPLIT` est envoyé avec un ensemble d'affectations partielles. Puis est envoyé le message
  - `MSG_STRING` plus "no change"
  - ou `MSG_QUANT` avec la formule partiellement traitée.

Notre fonction heuristique pour estimer la difficulté d'une tâche est très simple :

- Si une tâche a déjà été découpée par cette heuristique, elle est abordable,
- sinon, si le nombre de symboles propositionnels libres est inférieur à une constante, la tâche est abordable,
- sinon la tâche n'est pas abordable.

De même notre procédure de découpage sémantique est très simple. A la lecture de la formule, les symboles propositionnels libres sont empilés. Si un découpage est nécessaire, on calcule le nombre de symboles propositionnels libres à affecter sans toutefois dépasser une autre constante définissant le plus fin découpage autorisé. Ensuite, est dépilé le nombre de symboles propositionnels et est généré l'ensemble des affectations partielles possibles. Cet ensemble sera envoyé après `MSG_SPLIT`.

Les plus grandes améliorations que l'on puisse apporter au client sont au niveau de la fonction heuristique qui évalue la difficulté d'une tâche et la façon de choisir les symboles propositionnels du découpage sémantique. L'avantage de notre implantation est le comportement déterministe de ces deux composants, qui permet de limiter le nombre de variations lors de l'exécution. Ainsi, un client découpe toujours une même tâche de la même manière.

Actuellement, un client finit toujours un travail de découpage en retournant `MSG_STRING` suivi de "no change", mais dans le futur, en couplant une estimation de la difficulté et une résolution partielle, nous envisageons de pouvoir retourner des affectations partielles accompagnées du message `MSG_QUANT` suivies par la formule partiellement traitée.

5.3 Choix techniques

Une structure de données composite pour représenter les QBF. Afin de s'affranchir des problèmes liées à la mise sous forme prénexe et à la mise sous forme normale conjonctive, notre procédure devra travailler sur des QBF non prénexes non FNC closes. De plus nous voulons pouvoir traiter les implications, les bi-implications et les ou-exclusifs. Nous avons choisi comme format d'entrée QBF1.0 que nous avons étendu avec des symboles pour l'implication, la bi-implication et le ou-exclusif.

Notre structure de données n'a pas pour objectif d'être performante ou économe en espace. Nous cherchons un maximum d'expressivité et d'extensibilité. Afin de préserver l'information présente dans la formulation d'origine, une QBF est représentée par un ensemble d'éléments abstraits de formule, liés les uns aux autres sous forme d'un arbre. Un élément abstrait de formule impose uniquement deux actions aux éléments concrets : savoir démarrer un visiteur abstrait et savoir se sérialiser/désérialiser. Parmi les éléments concrets d'une formule, il y a les nœuds de l'arbre : les opérateurs logiques (binaires et unaires) et les quantificateurs. Puis il y a les feuilles de l'arbre : les littéraux et les constantes propositionnelles. Notre structure de données est donc un ensemble d'éléments composites héritant tous de l'élément abstrait de formule. Les visiteurs sont donc des traitements externes qui s'adaptent automatiquement à l'élément concret rencontré (design pattern composite + visitor) [11].

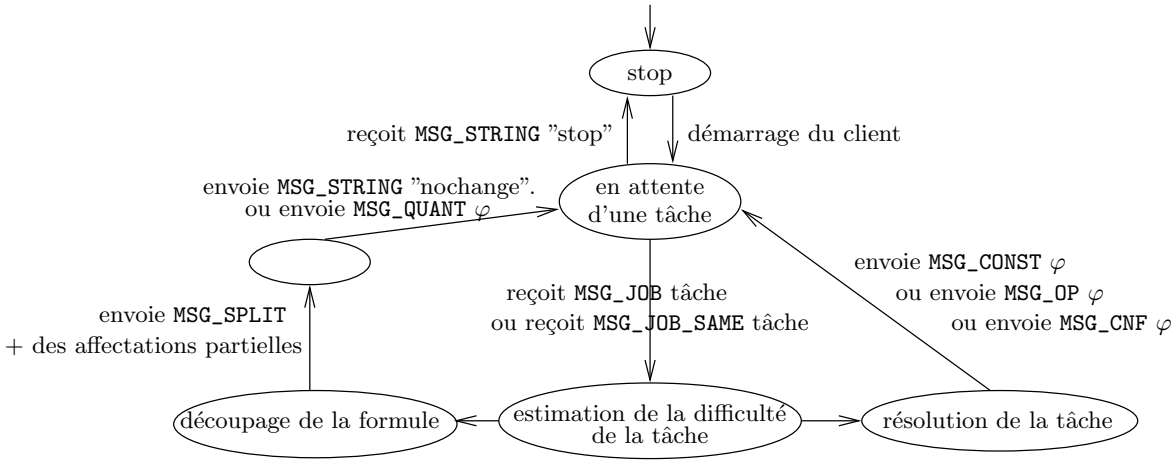

FIG. 2 – Automate du client

Une formule concrète est sérialisable afin de la transformer en flux, puis désérialisable afin de la restaurer en mémoire. Afin de stocker plus efficacement les formules possédant une partie sous forme normale conjonctive, nous avons défini un élément concret feuille FNC, stockant directement une matrice de littéraux. Il est alors facile de représenter des formules QBF sous FNC à l'aide uniquement des nœuds de quantificateurs et d'une feuille FNC.

MPI. Toute la partie communication est réalisée à l'aide du standard Message Passing Interface (MPI) [26]. Nous avons choisi comme implantation Open MPI. Pour notre approche avec une structure de données composite, MPI possède une lacune : il n'est pas possible simplement d'envoyer et recevoir des types de données complexes de façon native. La bibliothèque C++ Boost.MPI est une sur-couche répondant à cette contrainte à l'aide de Boost.Serialization. Toutes nos communications sont synchrones et utilisent le send/recv de Boost.MPI, faisant appel à MPI_Send et à MPI_Recv d'Open MPI.

MiniSat. Pour utiliser la méthode de la section 5, il faut pourvoir répondre à deux questions : la formule est-elle une tautologie ? Sinon, la formule est-elle insatisfiable ? Pour répondre, nous avons intégré la procédure de décision de `MiniSat`. Outre le fait que cette procédure soit très performante, ses sources sont disponibles et distribuées librement sous licence MIT. Les structures de données dans `MiniSat` sont très différentes des nôtres et sont orientées pour la performance. Nous avons donc implémenté une interface (un visiteur) permettant de formuler nos requêtes dans le format de données de `MiniSat`. A l'inverse, nous pouvons interpréter les réponses de `MiniSat` pour les intégrer directement dans notre structure de données. Il est ainsi possible à partir de notre structure de données expressive et extensible, de faire des traitements dans un modèle performant, le tout à l'aide de traitement en temps polynomial (linéaire) par rapport à la taille de la formule (nombre de noeuds).

Le caching. Afin de réduire le besoin en bande passante, chaque nœud de calcul garde en cache l'objet solveur Minisat de la dernière sous-formule traitée. Si une nouvelle tâche sur la même sous-formule arrive, le premier gain, est l'économie du transfert de la formule. Mais ce n'est pas tout : Minisat utilise le clause learning. L'instance déjà entraînée de Minisat peut possiblement répondre plus vite à une nouvelle question. L'idéal serait que les différents nœuds de calcul partage l'apprentissage réalisé par chacun, mais ce n'est pas le cas pour l'instant. De plus, il faudrait étudier en contre partie, le surplus de trafic généré par l'actualisation de ces informations.

5.4 Résultats expérimentaux

Afin d'évaluer notre architecture, nous avons effectué quelques tests préliminaires sur quelques instances de *qbflib* [13] et sur quelques problèmes que nous avons générés. Le découpage sémantique est strictement identique quel que soit le nombre de clients de calcul. Les tests ont été réalisés sur une machine hautes performances (HPC) composée de 12 serveurs de calcul Bull Novascale R422, connectés entre eux par 2 réseaux Ethernet Gigabit. Chaque serveur possède 2 nœuds de calcul de type 2x Intel(R) Xeon(R) E5440 à 2.83GHz et possède 16Go de mémoire locale. Les

	c8_8 ($\exists^{72}\forall^{64}$)		c8_16 ($\exists^{136}\forall^{128}$)		r4_5 ($\exists^{42}\forall^{35}$)		s5_4 ($\exists^{70}\forall^{56}$)	
1	105	1	7264	1	18483	1	58812	1
2	52	2.0	6627	1.1	11215	1.6	27741	2.1
4	28	3.8	4560	1.6	3601	5.1	19938	2.9
8	15	7.0	2437	3.0	4940	3.7	5537	10.6
16	11	9.5	3247	2.2	2268	8.1	4320	13.6
32	7	15.0	967	7.5	857	21.6	1027	57.2
64	6	17.5	3950	1.8	247	74.8	582	101.1
128	7	15.0	3548	2.0	128	144.4	780	75.4

TAB. 1 – Résultats obtenus sans prétraitement

machines tournent avec une version 2.6.18 du noyau Linux en version 64 bits.

Nous avons choisi quelques résultats pour illustrer différents cas. La première colonne de nos tables représente le nombre de clients de calcul auxquels il faut ajouter le processus maître tournant sur un processeur dédié. Les résultats sont sous la forme d'un couple *(temps en secondes/accélération)*.

La table 1 présente les résultats de 4 instances : counter8_8, counter8_16, ring4_5 et semaphore5_4. Elles font partie de la suite QBF1.0 disponible sur *qbflib*. Toutes ces formules sont prénexes, leur largeur syntaxique maximale est donc 1. Le seul choix que nous ayons pour traiter ces formules en parallèle est l'utilisation de l'extraction sémantique de sous-problèmes décrite dans la sous-section 5.2. Le premier problème, counter8_8, est simple ; l'augmentation du nombre de clients est vraiment efficace jusqu'à 8. L'explication est simple : plus nous utilisons de processus de calcul, plus le temps d'initialisation lié à MPI est long. Par exemple, l'initialisation pour 128 processus prend 5 secondes.

Pour évaluer notre approche avec beaucoup de ressources de calcul, il nous faut des problèmes plus conséquents. Le second problème, counter8_16, satisfait cette contrainte, malgré cela les temps sont irréguliers et l'accélération est assez mauvaise. Deux phénomènes surviennent ici. Tout d'abord, certains sous-problèmes sont très difficiles à résoudre. Par exemple, certaines tâches requièrent plusieurs centaines de secondes. Une tâche qui monopolise une ressource pendant 10% du temps, va limiter à 10 l'accélération maximale dans le meilleur des cas, c'est à dire en commençant la recherche par cette tâche. Le second phénomène, est lié à l'utilisation de `MiniSat` comme oracle. Chacun des 4 problèmes ne possède qu'une alternance de quantificateurs. C'est pourquoi, chaque nœud de calcul reçoit la formule une seule fois et crée une unique instance de la procédure `MiniSat` Toutes les autres tâches utiliseront cette instance en cache et tireront des bénéfices de l'apprentissage de clauses déjà réalisé localement lors des tâches précédentes. Le propre de notre exécution parallèle, c'est l'impossibilité de prévoir la distribution des tâches. Par extension, chaque sous-problème sémantique entraîne une instance de `MiniSat` et cet apprentissage est imprévisible et différent à chaque exécution. Plus le nombre de nœuds de calcul augmente, plus la probabilité d'apprendre localement une information intéressante avant un sous-problème difficile diminue. Avoir plus de nœuds de calcul, n'implique pas nécessairement une résolution plus rapide s'il n'est pas possible de partager des informations.

Pour le problème ring4_5, nous observons une accélération super-linéaire avec 64 et 128 processus de calcul. Comme pour le problème précédent, nous pensons que l'apprentissage de `MiniSat` a un effet, positif cette fois ci. Il est possible que certains sous-problèmes entraînent efficacement la plupart des instances de `MiniSat` Avec l'accroissement du nombre de nœuds de calcul, chaque solveur reçoit moins de sous-problèmes à résoudre. Peut-être cela leur permet-il de garder plus longtemps des informations plus pertinentes. Nous remarquons aussi qu'aucune tâche ne monopolise beaucoup de ressources. De part la nature aléatoire des exécutions parallèles, il faudrait multiplier les exécutions pour consolider nos hypothèses.

Nous observons une accélération super-linéaire pour le problème semaphore5_4 avec 32 et 64 clients. par contre, pour 128 le gain est inférieur à celui pour 64. Contrairement à ring5_4, certaines tâches sont très longues et peuvent représenter plus de 50% du temps total d'exécution pour 64 clients ou plus. Comme pour counter8_16, plus l'apprentissage est réparti sur différents clients plus les tâches longues sont pénalisantes.

La table 2 présente les résultats pour les mêmes problèmes mais avec l'application de quelques optimisations sur la formule en entrée. Le nœud maître, après lecture de la formule, applique récursivement une propagation naïve comparable à la propagation unitaire et cherche les littéraux monotones (n'apparaissant que dans une seule polarité). Ces améliorations simples montrent qu'il sera possible d'améliorer les performances de notre procédure de résolution en appliquant les techniques de l'état de l'art. Ces améliorations pourront être appliquées aussi sur les

	c8_8 ($\exists^{72}\forall^{64}$)		c8_16 ($\exists^{136}\forall^{128}$)		r4_5 ($\exists^{42}\forall^{35}$)		s5_4 ($\exists^{70}\forall^{56}$)	
1	52	1	6069	1	8718	1	25647	1
2	27	1.9	2959	2.1	3536	2.5	6530	3.9
4	14	3.7	1855	3.3	1597	5.4	1855	13.8
8	10	5.2	3338	1.8	914	9.5	2226	11.5
16	6	8.7	1114	5.4	349	25.0	1993	12.9
32	5	10.4	1527	4.0	202	43.2	778	33.0
64	5	10.4	1249	4.9	89	98.0	240	106.9
128	5	10.4	1750	3.5	38	229.4	588	43.6

TAB. 2 – Résultats obtenus avec un prétraitement

	adder_6 (non prénexe)		chaine_30 (non prénexe)	
1	3479	1	68343	1
2	1377	2.5	33068	2.1
4	995	3.5	16181	4.2
8	1788	1.9	7504	9.1
16	708	4.9	3780	18.1
32	1242	2.8	2043	33.5
64	1238	2.8	1023	66.8
128	1007	3.5	438	156.0

TAB. 3 – Résultats pour adder_6 et chaine_30

sous-problèmes. Nous relevons que pour ring4_5 les temps obtenus correspondent à une accélération super-linéaire et pour 128 clients le gain est de 229,4.

Contrairement aux problèmes précédents, adder_6 et chaine_30 ne sont pas prénexes et possèdent des bi-implications et/ou des ou exclusifs. Le problème adder_6 possède une largeur syntaxique de 2, sauf lors de la dernière itération. Le problème chaine_30 possède une largeur syntaxique de 30 lors de la première itération, puis de 1 ensuite. La table 3 résume les différents résultats. Pour adder_6, le gain est bon jusqu'à 4 clients. Pour ce problème la dernière itération est la tâche la plus longue, or cette tâche ne possède aucun symbole propositionnel libre : seul un client est actif. Pour chaine_30, nous observons une accélération super-linéaire. Dans un premier temps, le nœud maître distribue 30 sous-problèmes qui sont simplifiés très rapidement, puis la largeur syntaxique passe à 1. L'extraction sémantique de sous-problèmes prend le relais. Comme constaté pour ring5_4, les tâches sont de longueurs régulières.

6 Conclusion

Nous avons présenté dans cet article une nouvelle architecture pour la parallélisation du problème de validité des QBF dite « architecture de parallélisation syntaxique » par opposition aux architectures de parallélisation basée sur la sémantique des quantificateurs. Nous avons choisi d'implanter notre architecture selon un modèle maître/esclave avec pour oracle la procédure de décision `QSAT`. Cette implantation est déjà opérationnelle et la seule à notre connaissance à traiter en parallèle des QBF non FNC non prénexes. Le cadre proposé est suffisamment général pour intégrer d'autres procédures du moment même où elles réalisent une élimination des quantificateurs d'une QBF à symboles propositionnels libres.

Références

[1] G. Audemard and L. Sais. A Symbolic Search Based Approach for Quantified Boolean Formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.

[2] A. Ayari and D. Basin. Qubos : Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Formal Methods in Computer-Aided Design, Fourth International Conference, FMCAD 2002*. Springer-Verlag, 2002.

[3] M. Benedetti. skizzo : a suite to evaluate and certify QBFs. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, pages 369–376, 2005.

[4] M. Benedetti and H. Mangassarian. Experience and Perspectives in QBF-Based Formal Verification. *Journal on Satisfiability, Boolean Modeling and Computation*, 2008.

[5] A. Biere. Resolve and Expand. In *Proceedings of the 7th International Confrerence on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 59–70, 2004.

[6] M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In *Proceedings of the 5th Conference of the Italian Association for Artificial Intelligence (AIIA'97)*, pages 207–218, 1997.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5, 1962.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :201–215, July 1960.

[9] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 477–481, 2006.

[10] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI'00) and 12th Conference on Innovative Applications of Artificial Intelligence (IAAI'00)*, 2000.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Assison-Wesley Professional Computing Series, 1994.

[12] I.P. Gent, H.H. Hoos, A.G.D. Rowley, and K. Smyth. Unsing Stochastic Local Search to Solve Quantified Boolean Formulae. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, 2003.

[13] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.

[14] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE : A System for Deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 364–369, 2001.

[15] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1) :12–18, 1995.

[16] H. Kleine Büning and X. Zhao. On Models for Quantified Boolean Formulas. In *Logic versus Approximation, In Lecture Notes in Computer Science 3075*, 2004.

[17] M. Lewis, P. Marin, T. Schubert, M. Narizzano, B. Becker, and E. Giunchiglia. PaQuBE : Distributed QBF Solving with Advanced Knowledge Sharing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pages 509–523, 2009.

[18] M. Lewis, T. Schubert, and B. Becker. QMiraXT - A Multithreaded QBF Solver. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'09)*, pages 7–16, 2009.

[19] F. Lonsing and A. Biere. Nenofex : Expanding NNF for QBF Solving. In *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, pages 196–210, 2008.

[20] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT'72)*, pages 125–129, 1972.

[21] D.A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130 :291–328, 2003.

[22] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10 :323–352, 1999.

[23] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1192–1197, 1999.

[24] T. Schubert, M. Lewis, and B. Becker. PaMiraXT : Parallel SAT Solving with Threads and Message Passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :203–222, 2009.

[25] B. Selman, H.A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1993.

[26] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, 1995.

[27] I. Stéphan, B. Da Mota, and P. Nicolas. From (Quantified) Boolean Formulas to Answer Set Programming. *Journal of logic and computation*, 19(4) :565–590, 2009.

[28] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3 :1–22, 1977.

[29] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *International Conference on Computer Aided Design (ICCAD'02)*, 2002.