SPIKEpar : une interface parallèle du démonstrateur automatique SPIKE

Sorin Stratulat

SPIKEpar : une interface parallèle du démonstrateur automatique SPIKE

Sorin Stratulat

LORIA, INRIA
B.P. 239
54506 VANDŒUVRE-LES-NANCY CEDEX, FRANCE

Résumé

Nous décrivons une interface parallèle pour le démonstrateur automatique SPIKE telle qu'elle est mise en œuvre sur un réseau de stations de travail monoprocesseur. L'approche maître-esclave abordée pour paralléliser le calcul symbolique lié aux preuves mécaniques par récurrence permet de partager la quantité de travail entre les processeurs selon le modèle de parallélisation de type ET, conforme à la stratégie. Un processus SPIKE effectue des calculs à gros grain au niveau des clauses.

Mots-clés : calcul symbolique parallèle, systèmes de réécriture, démonstration de théorèmes par récurrence, SPIKE, parallélisme à gros grain

1. Introduction

La motivation à ce travail est la programmation des environnements parallèles et distribués pour faire du calcul symbolique correct et rapide. Notre but est de voir jusqu'à quel niveau les preuves par récurrence basées sur la réécriture peuvent gagner en efficacité grâce à leur parallélisation et, en même temps, de trouver des méthodes adéquates pour les mettre en œuvre.

Les preuves par récurrence basées sur la réécriture sont connues comme étant difficiles à paralléliser. Une des raisons est la forte dépendance de la stratégie de preuve choisie, elle-même étant sensible aux décisions de programmation dynamique des tâches entre processeurs. La stratégie utilisée pendant une preuve influence beaucoup les performances du démonstrateur. Elle se répercute dans les tests finaux de performance et rend difficile l'analyse de l'efficacité du travail en parallèle.

Il y a deux approches pour paralléliser des programmes liés au calcul symbolique. La première utilise le *parallélisme de l'espace de recherche*, aussi nommé *parallélisme de type OU*, qui consiste à trouver parmi plusieurs stratégies celle qui mène à une solution. Le gain en performance peut être obtenu par l'exécution parallèle de différentes stratégies jusqu'à obtenir une première solution. Mais, si dans l'application une stratégie optimale est employée, les avantages à utiliser cette approche sont perdus.

La deuxième approche exploite le *parallélisme de travail*, aussi connu comme *parallélisme de type ET*, et s'appuie sur le partage du travail global entre plusieurs processeurs. Dans sa version pure, l'algorithme parallèle s'exécute chaque fois avec la même stratégie ; on parle alors d'une *parallélisation conforme à la stratégie* [1]. Une telle approche a de nombreux avantages, comme la possibilité d'effectuer des calculs déterminés et prévus, ainsi que des expériences reproductibles.

La grande complexité d'une preuve par récurrence basée sur la réécriture permet des parallélisations à plusieurs niveaux de granularité. La plupart des travaux théoriques sont consacrés à la parallélisation à grain fin (p.e. le traitement du filtrage parallèle), tandis que les expérimentations sont focalisées plutôt sur l'autre extrémité, cas où l'efficacité du calcul est facile à obtenir mais les effets de la stratégie utilisée peuvent intervenir.

Le but de cet article est la description de *SPIKEpar*, une interface parallèle de SPIKE [3] qui combine le schéma maître-esclave avec un schéma de parallélisation à gros grain et conforme à la stratégie.

2. Réécriture des termes et preuve par récurrence avec SPIKE

Dans cette section nous décrivons l'opération de réécriture et les concepts fondamentaux liés à une preuve par récurrence basée sur la réécriture à un niveau suffisant pour une compréhension de la méthode de parallélisation. Un aperçu plus détaillé des preuves par réécriture des termes se trouve en [4].

Un *terme* est construit par des constantes, variables et symboles de fonction, de la manière usuelle. Les opérations de base sur les termes sont des instantiations, des tests d'égalité structurelle et des filtrages. Un terme t' est une *instance* de t s'il peut être obtenu en remplaçant les variables de t par des termes ; on écrit $t' = t\sigma$, où σ représente la substitution utilisée à l'instantiation. Un terme s *filtre* un autre terme t s'il existe une substitution θ telle que $s\theta = t$.

On associe à chaque terme une *sorte*, qui dénote un ensemble de valeurs, lui-même caractérisé par une description finie appelée *ensemble couvrant*. $s = t$ est une *équation* si s et t sont des termes de même sorte. Un *littéral* est une équation ou sa négation. Une disjonction de littéraux constitue une *clause*.

Une *règle de réécriture non-conditionnelle* est notée $l \to r$, où l et r sont des termes. Elle peut être utilisée pour réduire un terme t s'il contient une instance l' de l qui est remplacée par l'instance correspondante de r. Un ensemble de règles de réécriture forme un *système de réécriture de termes (SRT)*. Un SRT $\mathcal{R}$ représente une relation de réduction telle que $s \to_{\mathcal{R}} t$ s'il y a une règle de $\mathcal{R}$ qui réduit s en t. La relation de réduction $\to_{\mathcal{R}}$ induit un ordre sur les termes et doit avoir de bonnes propriétés, par exemple la terminaison. L'ordre sur les termes peut s'étendre à un ordre $\preceq$ sur les clauses.

Étant donné un ensemble de clauses Ax (nommées *axiomes*) et une clause ϕ (nommée *conjecture*), on dit que ϕ est une *conséquence logique* de Ax, notée $Ax \models \phi$, si ϕ est valide dans tous les modèles de Ax. Si pour toute substitution *close* γ (i.e. les termes remplaçants sont sans variable) on a $Ax \models \phi\gamma$, on dit que ϕ est une *conséquence inductive* de Ax et on la note par $Ax \models_{ind} \phi$.

Généralement, une preuve mécanisée avec un démonstrateur automatique émerge de l'application étape par étape de *règles d'inférence* de son système d'inférence. Le système d'inférence de SPIKE consiste en deux classes de règles d'inférence (voir à la Figure 1) : des règles de type *Generate* et *Simplify*, conf. [6].

Une règle de type *Generate* instancie certaines variables de la conjecture traitée avec des éléments de l'ensemble couvrant, et puis applique une étape de simplification. Les substitutions créées s'appellent des *substitutions par des ensembles couvrants (s.e.c.)*. Une instance d'une règle de type *Simplify* peut transformer une clause en plusieurs clauses. Une stratégie de preuve standard de SPIKE consiste en l'application, dans une boucle infinie, de l'opération de normalisation des conjectures qui est obtenue après une suite d'étapes de simplification, suivie d'une étape de récurrence. Le *niveau de récurrence* d'une clause indique le nombre minimal d'étapes de récurrence nécessaires pour sa création.

Generate: $(E \cup \{C\}, H) \vdash (E \cup E', H \cup \{C\})$
si pour toute s.e.c. σ de C,
$Ax \models_{ind} (E \cup E' \cup H \cup \{C\})_{\prec C\sigma} \Rightarrow C\sigma$.

Simplify: $(E \cup \{C\}, H) \vdash (E \cup E', H)$
si $Ax \models_{ind} (E \cup E' \cup H)_{\preceq C} \Rightarrow C$

FIG. 1 – *Le système d'inférence générique*

L'application d'une règle d'inférence est vue comme une transition, notée $\vdash$, d'une paire d'ensembles de clauses (E_i, H_i) en une nouvelle paire (E_{i+1}, H_{i+1}), où E_i et E_{i+1} sont des ensembles de conjectures, H_i et H_{i+1} des ensembles d'hypothèses de récurrence. Une suite d'applications des règles d'inférence, appelée *dérivation*, finit avec *succès* si la dernière paire de la dérivation est de la forme $(\emptyset, H)$. Une stratégie est *équitable* si l'ensemble de conjectures persistantes $(\cup_i \cap_{j \geq i} E_j)$ de chacune de ses dérivations est vide.

3. Parallélisation du prouveur

L'algorithme utilisé dans la mise en œuvre de *SPIKEpar* a été conçu pour exploiter le parallélisme à gros grain sur une architecture de processeurs à mémoire distribuée en suivant la paradigme *diviser pour régner* lors de l'attribution de travail indépendant aux différents processus.

L'analyse du système d'inférence de SPIKE, avec une stratégie plus raffinée qui interdit les simplifications d'une conjecture avec d'autres conjectures, nous a permis d'isoler des fragments de preuve qui peuvent être exécutés indépendamment. Une première étape de parallélisation est basée sur l'observation du fait que chaque conjecture de l'ensemble de conjectures intermédiaires, issues d'une étape de récurrence, est candidate à une étape de normalisation. De plus, un ensemble de conjectures est valide si et seulement si chaque conjecture, considérée séparément, est valide. Par conséquent, l'opération de

normalisation peut s'exécuter indépendamment pour chaque conjecture à l'aide d'un processus SPIKE lancé avec un même environnement de preuve. Dès que les opérations de normalisation sont terminées, on joint les conjectures résultat dans un nouvel environnement global de preuve, puis on tue les processus SPIKE.

1. $(E, H) \leftarrow (E_0, H_0)$; fin_preuve $\leftarrow$ faux;
2. tant que $\neg$ fin_preuve $\wedge$ $E \neq \emptyset$ **exécute**
% (Simplif. parallèle) E est de la forme $\cup_{j=1}^{n}\{C_j\}$
3. processus k : $(\{C_k\}, H) \vdash^{*Simp.} (E'_k, H)$
4. (jointure) $(E, H) \leftarrow (\cup_{j=1}^{n} E_j, H)$
% (Récurrence parallèle) E est de la forme $\cup_{j=1}^{m}\{D_j\}$
5. processus k : $(\{D_k\}, H) \vdash^{Gen.} (E'_k, H \cup \{D_k\})$
6. (jointure) $(E, H) \leftarrow (\cup_{j=1}^{m} E'_j, H \cup E)$
7. fin boucle

FIG. 2 – *L'algorithme mis en œuvre par SPIKEpar*

De la même manière on raffine encore plus la stratégie en interdisant, pendant une opération de récurrence, la simplification avec des hypothèses de récurrence d'un même niveau de récurrence afin de permettre une opération de récurrence parallèle. Ainsi, une opération de récurrence globale peut être considérée comme la composition des opérations de récurrence sur chaque conjecture. Les conclusions de récurrence et les hypothèses de récurrence, calculées par des processus SPIKE, sont réunis pour former le nouvel environnement de la preuve. L'algorithme de parallélisation proposé est présenté dans la Figure 2. Notons qu'un échec de la preuve d'une conjecture issue d'un processus SPIKE est un échec global.

La preuve de correction des résultats issus de *SPIKEpar* est basée sur la correction d'une preuve de SPIKE. Pour une application donnée il y a des points de synchronisation de toutes les traces de preuves générées avec un nombre arbitraire de processeurs, supposant qu'un processeur exécute un seul processus à la fois. Par exemple, l'ensemble (E, H) calculé aux pas 4 (ou 6) est identique dans toutes les preuves, comptant pour la même itération. Par conséquent, le travail fait avec plusieurs processeurs est exactement le même que celui effectué en utilisant une stratégie séquentielle, sur un seul processeur. La preuve de correction s'avère triviale et s'appuie sur l'observation que toute preuve générée avec une stratégie séquentielle peut se rejouer avec SPIKE.

Nous avons aussi montré que le système d'inférence construit avec les règles de simplification et récurrence parallèle est *réfutationnellement correct* (i.e. le système transforme des prémisses valides en des conséquences valides). L'algorithme présenté définit une stratégie équitable parce que toutes les conjectures de l'ensemble E des opérations de simplification et récurrence paralléle sont traitées. De plus, il induit une parallélisation conforme à la stratégie qui garantit la même stratégie de preuve inchangeable vis-à-vis de la programmation des tâches ou du nombre de processeurs employés.

4. Mise en œuvre

La mise en œuvre de *SPIKEpar* suit un schéma maître-esclave qui est constitué d'un processus maître et un ou plusieurs processus esclave. Le processus maître exécute l'algorithme de la Figure 2 ; il identifie les fragments de preuve qui peuvent être traités indépendamment, crée des processus esclave particuliers au type d'opération en cours d'exécution, collecte et interprète les résultats des processus esclave.

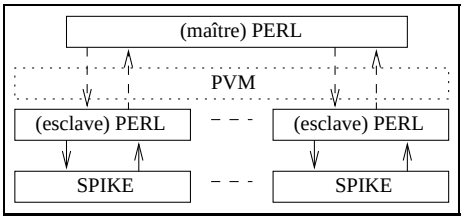

FIG. 3 – *Le schéma maître-esclave*

Dans la Figure 3, chaque processus esclave contrôle un processus SPIKE afin de traiter la conjecture attribuée par le processus maître et veille à l'exécution correcte de l'opération qu'il a prise en charge. Le transfert des paramètres d'un processus esclave à un processus SPIKE se fait à l'aide des fichiers. Lorsque la preuve de la conjecture est lancée, le processus esclave maîtrise son état courant par l'analyse des mots-clés de la trace de preuve générée par le processus SPIKE. Le résultat de l'opération est envoyé au processus maître dès qu'il est détecté dans la trace de la preuve. Si un processus esclave signale un échec, le processus maître arrête tous les autres processus esclave et termine avec un échec global. La création/destruction des processus esclave et le transfert des paramètres entre un processus esclave et maître se fait à l'aide de PVM [5]. L'utilisation de PVM permet également le chargement automatique des processus sur les processeurs disponibles dans sa configuration. La communication entre processus via PVM est faible et a une influence négligeable sur les temps d'exécution ; elle n'apparaît qu'au moment de la création et vers la fin du processus esclave.

entrée	t. exécution sur n processeurs (s)				n° processus esclave	
	$n=1$	$n=2$	$n=3$	$n=4$	Generate	Simplify
`sorted`	23	17	14	13	7	20
`gct`	258	205	183	153	59	106

TAB. 1 – *Les résultats*

L'algorithme de parallélisation et les codes des processus esclave ont été écrits en PERL [7]. Il nous permet d'effectuer des analyses efficaces des traces de preuves. Le code PERL de *SPIKEpar* s'étend sur trois fichiers : un est associé au processus maître et deux à chaque type de processus esclave, selon la nature de l'opération effectuée. Le nombre de lignes de code est d'environ 1000.

5. Résultats expérimentaux

Nous avons mis en œuvre et testé *SPIKEpar* sur un réseau de 4 stations de travail DEC/ALPHA, de type 500/333, avec une mémoire vive de 128 Moctets.

Un certain nombre de problèmes a été traité avec *SPIKEpar*, dont le problème du tour de cartes de Gilbreath (`gct`) et la correction d'un algorithme de tri par insertion (`sorted`) [2]. Les résultats sont affichés dans la Table 1. A noter que les résultats confirment que la parallélisation est *échelonnable*, dans le sens où, si le problème contient assez de parallélisme, les temps d'exécution s'améliorent avec l'ajout de processeurs. Nous avons également montré cette propriété théoriquement.

Pour le problème `gct`, l'efficacité obtenue en utilisant deux processeurs est plus élevée par rapport aux cas où $n > 2$. Cela s'explique en analysant la structure de la preuve ; les opérations de récurrence sur les deux conjectures initiales demandent un temps important rapporté au temps total de la preuve. Ainsi, l'ajout d'un nouveau processeur ne contribuera plus à la construction de ce fragment de preuve.

6. Discussions

Les temps d'exécution d'une preuve avec *SPIKEpar* dépendent aussi de la performance de SPIKE. Une version de SPIKE plus performante va induire une version de *SPIKEpar* plus efficace. Nous pensons que *SPIKEpar* est facile à adapter à une nouvelle version de SPIKE, vu qu'en principe il suffit d'adapter aux nouveaux mots-clés les expressions régulières qui reconnaissent les mots-clés des traces de preuve.

Il y a également des inconvénients, par exemple, pour le passage à une parallélisation de granularité plus fine ; la parallélisation des opérations sur des termes ne peut plus être contrôlée avec *SPIKEpar*. La solution, qui constitue un projet futur, est la conception d'une version parallèle de SPIKE. Elle permettra, entre autres, l'accès direct aux structures de données et un contrôle plus précis et efficace de la preuve.

Bibliographie

1. Bündgen (R.), Göbel (M.) et Küchlin (W.). – Strategy compliant multi-threaded term completion. *Journal of Symbolic Computation*, 1996.
2. Bouhoula (A.). – *Preuves Automatiques par Récurrence dans les Théories Conditionnelles.* – Thèse de PhD, Université Nancy I, mars 1994.
3. Bouhoula (A.) et Rusinowitch (M.). – Implicit induction in conditional theories. *Journal of Automated Reasoning*, vol. 14, n2, 1995, pp. 189–235.
4. Dershowitz (N.) et Jouannaud (J.P.). – *Rewrite systems*, chap. 6, pp. 243–320. – North-Holland, Amsterdam, 1990, *Handbook of Theoretical Computer Science*, volume B:Formal methods and Semantics.
5. Geist (A.) et al. – *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* – MIT Press, 1994.
6. Naidich (D.). – *On Generic Representation of Implicit Induction Procedures.* – Rapport technique nCS-R9620, CWI, 1996.
7. Wall (L.) et al. – *Programming Perl.* – O'Reilly, 1997, $2^{\text{ième}}$ édition.