Construction parallèle de l'automate *LR(0)* : Une application de Multilisp à la compilation

Dominique Boucher et Marc Feeley

Département d'Informatique et R. O. / Université de Montréal
{boucherd,feeley}@iro.umontreal.ca

1 Introduction

Nous nous intéressons ici à la parallélisation des générateurs d'analyseurs syntaxiques *LALR(1)*, tels YACC et Bison, et à leur implantation en Multilisp [Halstead, 1985], un dialecte parallèle de Scheme, dont le choix est motivé par la nature symbolique du problème. La construction de tels analyseurs requière habituellement trois phases distinctes [Aho, Sethi et Ullman, 1986] :

1. la construction de l'automate *LR(0)*,
2. le calcul des contextes de réductions, aussi appelés ensembles de prévision, et
3. la construction des tables d'analyse.

Un algorithme parallèle permettant d'effectuer le calcul des deux dernières phases est décrit dans [Bermudez et al., 1990]. Le facteur d'accélération théorique de cet algorithme est de $\min(N, T)$, où T est le nombre de terminaux de la grammaire et N est le nombre de processeurs. Or la construction de l'automate *LR(0)* représente typiquement entre 25 et 50% du temps total de calcul. Le tableau suivant donne les temps de calcul (en secondes) des trois phases pour 3 grammaires, telles que calculées séquentiellement par Bison sur un seul processeur du multiprocesseur à mémoire partagée BBN GP1000 [BBN, 1989] :

Grammaire	Phase 1 (en sec.)	Phases 2 et 3 (en sec.)	% Phase 1
Ada	2.25	2.67	45.7
C	1.33	3.06	30.3
C++	6.18	16.54	27.2

Dans Bison, la phase 1 est calculée d'une manière standard (décrite plus bas) et les phases 2 et 3 sont calculées suivant la méthode décrite dans [DeRemer et Pennello, 1982].

C'est donc dire qu'une partie importante de la génération des tables *LALR* est susceptible d'être accélérée par une implantation parallèle efficace. Notre travail montre que l'automate *LR(0)* peut être construit efficacement en parallèle. Nous présentons l'algorithme que nous avons implanté et les problèmes de performance posés par Multilisp dans l'implantation d'applications symboliques parallèles.

2 Construction de l'automate *LR(0)* en parallèle

Les états de l'automate *LR(0)* d'une grammaire hors-contexte G sont des ensembles d'*items LR(0)* et les transitions entre états se font sur les catégories et les terminaux de G. Chaque

item est de la forme $A \to \alpha \cdot \beta$, où $A \to \alpha\beta$ est une règle de production de la grammaire. La *fermeture LR(0)* d'un ensemble d'items I est le plus petit ensemble F contenant I tel que si $A \to \alpha \cdot X\beta \in I$ et $X \to \gamma$ est une règle production de la grammaire, alors $X \to \cdot\gamma \in F$. L'état initial de l'automate est obtenu en prenant la fermeture de $\{S' \to \cdot S\}$ où S' est une catégorie différente de toutes les catégories de G et S est la catégorie principale de G. L'état d'arrivée d'une transition sur X dont l'état de départ est I se calcule en prenant la fermeture *LR(0)* de $I' = \{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\}$.

L'algorithme séquentiel standard [Aho, Sethi et Ullman, 1986] pour le calcul de cet automate procède état par état, en examinant une liste ne contenant initialement que l'état initial. Pour chaque état I de la liste, on détermine les transitions possibles qui partent de I et les états d'arrivée de ces transitions. Parmi ces états, ceux qui n'apparaissent pas dans la liste y sont ajoutés. Une simple observation de la construction permet de constater que toutes les transitions qui partent d'un état I ont des destinations différentes. Ceci suggère de traiter en parallèle tous les états directement accessibles à partir de l'état I. Il est donc théoriquement possible d'obtenir une accélération de $\min(P, C)$, où P est le nombre de processeurs et C est une constante qui dépend de la grammaire. Voici le pseudocode d'un algorithme parallèle résolvant le problème :

Procédure *TraiterÉtat(s:état)*
    Créer l'état s;
    X := Ensemble des états directements accessibles de s;
    pour tout $x \in X$ **en parallèle faire**
        si x n'est pas déjà créé, TraiterÉtat(x);

TraiterÉtat($\{ S' \to \cdot S \}$)

3 L'implantation en Multilisp

Nous avons implanté cet algorithme parallèle en Multilisp. Le compilateur Multilisp optimisant Gambit [Feeley, 1993a] a été utilisé à cette fin. Nous avons tout d'abord réalisé une version séquentielle en Scheme à partir du programme Bison (écrit en C). Pour la version parallèle, seuls quelques changements mineurs ont dû être apportés à la version séquentielle :

- Une table de hachage est ordinairement utilisée pour maintenir l'ensemble d'états de l'automate et vérifier rapidement si un état est déjà créé. Nous avons ajouté un verrou à chaque liste de collision pour empêcher que deux processeurs puisse modifier cette liste en même temps.
- Nous avons ajouté un verrou à la variable globale permettant d'assigner un entier unique à chaque nouvel état.
- Nous avons ajouté un appel à `future` dans la boucle qui traite les états directement accessibles, ainsi qu'un appel à `touch`. Ce sont les deux seules primitives de Multilisp permettant d'exprimer le parallélisme dans un programme. `Future` permet de créer une nouvelle tâche tandis que `touch` permet de synchroniser les tâches démarrées avec `future`. Avec Gambit, le placement des tâches se fait automatiquement en fonction de la charge grâce au mécanisme de "Lazy Task Creation" [Feeley, 1993b].
- Il n'y a plus de liste globale contenant les états déjà créés.

De plus, quelques variables globales ont été remplacées par des variables locales passées en paramètre aux fonctions appelées. Ces modifications, somme toute mineures, montrent à quel point Multilisp se prête bien à l'écriture d'applications parallèles. De plus, le programmeur n'a pas à se soucier de la répartition des tâches au moment de la conception car le système s'en charge à l'exécution.

En se limitant à ces quelques modifications, les résultats n'ont pas été très bons. Le problème, c'est que les structures de données nécessaires à la construction de l'automate sont calculées par un algorithme séquentiel, impliquant que toutes les données sont emmagasinées dans la mémoire d'un seul processeur, causant ainsi des problèmes de contention. Puisque Multilisp ne possède pas de primitive standard ou de mécanisme automatique pour répartir les données sur les différents processeurs, les processeurs essaient tous d'accéder à la même banque de mémoire en même temps, provoquant ainsi un goulot d'étranglement.

Pour enrayer le problème, il a fallu copier ces structures partagées dans la mémoire locale de chaque processeur. Malheureusement, cette opération n'est pas traitée de façon élégante par le *garbage-collector* (GC) de Gambit qui ne peut récupérer l'espace occupé par des structures copiées localement. Dans un système réaliste, il faudrait trouver une solution à ce problème. À part ce problème, l'algorithme de GC parallèle utilisé par Gambit a bien fonctionné.

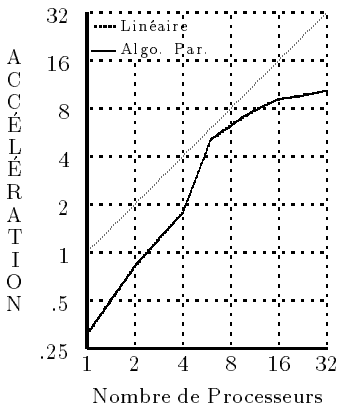
4 Résultats expérimentaux

Les tests ont été effectués sur un multiprocesseur GP1000 de BBN avec 1 à 32 processeurs. Une courbe d'accélération pour la grammaire de C est donnée à la figure 1. L'accélération est calculée avec la formule T_s/T_p, où T_s est le temps de l'implantation *séquentielle* en Scheme et T_p est le temps de l'implantation parallèle en Multilisp sur p processeurs. Les temps comprennent uniquement le calcul de l'automate *LR(0)*.

Deux remarques s'imposent toutefois. D'abord, on peut noter une certaine irrégularité dans la courbe d'accélération avec 4 processeurs et moins. La nature discrète du GC est à la source de cette irrégularité. Avec plus de 4 processeurs, le GC n'est plus appelé, ce qui diminue sensiblement le temps de calcul. On remarque également qu'il n'y a presque plus d'accélération au-delà de 16 processeurs. La quantité de parallélisme contenue dans l'automate pour la grammaire de C permet d'expliquer cela. La figure 2 donne le profil de parallélisme en fonction du temps pour des exécutions avec 8 et 16 processeurs. Les zones claires du profil donnent la proportion des processeurs qui effectuent une tâche utile et la zone noire indique des processeurs sans travail. Dans le profil sur 16 processeurs, on voit bien qu'on approche la limite du parallélisme contenu dans le problème.

5 Conclusion

Nous avons montré qu'une formulation parallèle de l'algorithme de construction de l'automate *LR(0)* permet d'obtenir des accélérations appréciables et que le langage Multilisp se prête bien à cette formulation. Par contre, ce langage offre peu de contrôle sur la répartition des données entre les processeurs et peu de contrôle sur l'emplacement des tâches. Il est donc difficile d'exploiter la localité de traitement. D'un autre côté, le balancement de la charge est facile puisque le système s'en occupe automatiquement. Nous comptons examiner des applications symboliques de plus grande envergure pour confirmer nos conclusions.



# de proc.	Temps en sec.	Accél.	# de GC
Séq.	3.23	—	0
1	10.42	0.31	5
2	3.89	0.83	3
4	1.78	1.81	1
6	0.63	5.12	0
8	0.51	6.32	0
10	0.44	7.33	0
12	0.40	8.06	0
14	0.37	8.71	0
16	0.35	9.22	0
20	0.34	9.49	0
32	0.31	10.41	0

Figure 1: Courbe d'accélération pour la grammaire de C

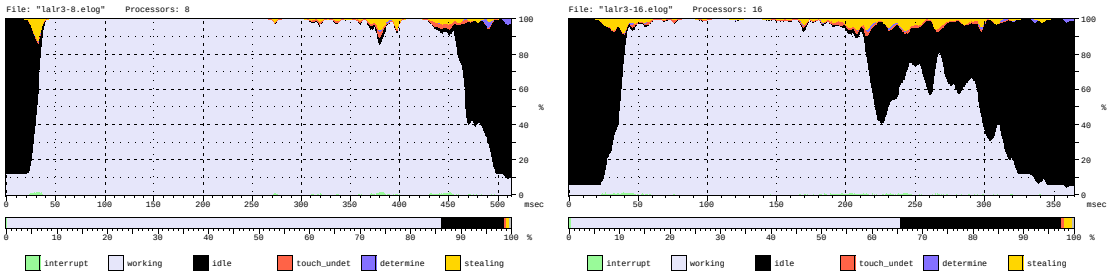


Figure 2: Utilisation des processeurs avec 8 et 16 processeurs

Références

[Aho, Sethi et Ullman, 1986] A. V. Aho, R. Sethi et J. D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BBN, 1989] BBN Advanced Computers Inc., Cambridge, MA. *Inside de GP1000 Computer,* 1989.

[Bermudez et al., 1990] M. E. Bermudez, R. Newman-Wolfe et G. Logothetis. Parallel Construction of *SLR(1)* and *LALR(1)* Parsers. *International Journal of Parallel Programming*, 19(3):163–184, 1990.

[DeRemer et Pennello, 1982] F. DeRemer et T. Pennello. Efficient Computation of *LALR(1)* Look-Ahead Sets. *ACM TOPLAS*, 4(4):615–649, Octobre 1982.

[Feeley, 1993a] M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* Thèse de PhD, Brandeis University Departement of Computer Science, 1993.

[Feeley, 1993b] M. Feeley. A Message Passing Implementation of Lazy Task Creation. Dans *Parallel Symbolic Computing: Languages, Systems, and Applications,* LNCS 748, édité par R. Halstead et T. Ito, novembre 1993, 14 pages.

[Halstead, 1985] R. Halstead. Multilisp : A language for concurrent symbolic computation. Dans *ACM Trans. on Prog. Languages and Systems,* pages 501–538, Octobre 1985.