Horloges structurelles pour la désynchronisation de programmes data-parallèles

Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf et Bernard Virot*

1. Introduction

Le modèle de programmation data-parallèle sait tirer partie de l'extensibilité des architectures parallèles en exprimant le parallélisme au moyen des structures de données qui sont promues des scalaires vers les vecteurs. Dans le cadre des architectures MIMD, le compilateur doit tendre à désynchroniser les éléments de calcul, sans perdre la sémantique synchrone du modèle initial.

Dans [4], Rice présente un langage de type data-parallèle dont le modèle d'exécution à base de processus CSP permet d'exploiter les architectures MIMD. Trop proche de l'aspect centralisé du modèle SIMD, cette approche exploite très peu l'asynchronisme de CSP.

Dans [2], Hatcher et Quinn proposent une approche systématique pour le langage DataParallel C : les opérateurs de composition séquentielle sont considérés comme des synchronisations globales. Afin d'optimiser, seules les synchronisations globales placées avant chaque communication sont conservées pour assurer que les informations échangées soient les mêmes que celles qui le seraient dans une interprétation purement synchrone. Le traitement consiste à transformer le programme pour extraire les synchronisations et communications des structures de contrôle, afin que tous les processeurs virtuels puissent participer à la synchronisation, quelle que soit leur activité.

Dans [3], Le Guyadec propose une autre approche en définissant un cadre formel qui permet de valider certaines optimisations du processus de compilation. Il introduit le langage $\mathcal{D}$ qui repose sur le modèle data-parallèle augmenté de structures de contrôle asynchrones permettant d'exprimer certaines indépendances entre processeurs virtuels. Un modèle d'exécution à mémoires jumelles réduit les dépendances de données en masquant certaines transitions locales. Il permet d'implanter des synchronisations partielles, c'est-à-dire celles impliquant seulement les indices actifs. L'interprétation d'une synchronisation partielle repose sur la notion de partition de l'ensemble des indices actifs. Chaque modification de cette partition implique une synchronisation. Par conséquent l'expression de la concurrence entre éléments de calcul est bornée par les structures de contrôle qui modifient l'ensemble des éléments actifs.

Dans cet article, nous exploitons le modèle d'exécution à mémoires jumelles ainsi que la notion de synchronisation partielle pour définir un langage data-parallèle asynchrone, appelé $\mathcal{SCL}$ (*Structural Clock Language*). Notre approche se distingue de la précédente par l'interprétation des synchronisations qui repose sur un mécanisme de datation d'événements. L'asynchronisme n'est plus borné par les structures de contrôle. Il offre le moyen d'exprimer plus de concurrence. Cette extension nécessite la définition d'un algorithme de synchronisation partielle.

Nous construisons notre algorithme sur des *événements de contrôle* (entrée ou sortie de structures de contrôle). Nous introduisons des *horloges structurelles* qui codent la position des processeurs virtuels vis-à-vis des *événements de contrôle*. Nous définissons un ordre sur ces horloges permettant d'aboutir à une interprétation correcte des synchronisations partielles.

2. Le langage $\mathcal{SCL}$

Dans cette section, nous rappelons les fondements du modèle data-parallèle ainsi que le modèle d'exécution à mémoires jumelles [3]. Nous donnons ensuite une présentation informelle du langage $\mathcal{SCL}$, sans structure itérative. Cette forme minimale de langage permet toutefois d'expliciter l'intérêt

*. LIFO - IIIA, Université d'Orléans, 4 rue Léonard De Vinci - BP 6759, F-45067 Orléans Cedex 02 - FRANCE. Contact Bernard.Virot@lifo.univ-orleans.fr

de cette approche sans l'obscurcir par les complications techniques dues aux boucles. Pour une description complète du langage, se reporter à [1].

2.1. Le modèle data-parallèle

Les objets de base du modèle data-parallèle sont des tableaux à accès parallèle appelés vecteurs. La composante d'une variable parallèle X localisée à l'indice u est notée $X|_u$. Deux catégories d'actions sont applicables aux objets : des opérations *locales*, et des *réarrangements* globaux. Un programme est une composition séquentielle de telles actions. A chaque action est associé l'ensemble des indices auxquels elle s'applique, i.e. l'ensemble des indices dits *actifs* (vis-à-vis de cette instruction) ; les autres indices sont dits *inactifs*. Les expressions légales sont les expressions *pures*, i.e. sans effets de bord. La valeur d'une expression pure à chaque indice u ne dépend que des valeurs des variables à cet indice. Nous utilisons un vecteur constant particulier, appelé $This$, dont la valeur à chaque indice u n'est autre que la valeur u elle-même : $This|_u = u$.

2.2. Un modèle d'exécution à mémoires jumelles

Nous conservons ici le modèle d'exécution à mémoires jumelles conçu pour le langage $\mathcal{D}$ [3] dans le but de réduire les dépendances de données. Il est basé sur un tableau de processeurs virtuels asynchrones. Chaque indice u possède une *mémoire privée* qui contient les données vectorielles situées à l'indice u, et une *mémoire publique* qui contient une copie de la mémoire privée du même indice. La mise à jour des mémoires publiques est réalisée au moyen d'une instruction spécifique : copy. Les mémoires privées sont dédiées aux calculs locaux, les mémoires publiques étant réservées aux communications. Pour chaque indice, les transitions effectuées entre l'exécution de deux instructions copy sont masquées pour les autres indices. Ceci permet d'exprimer plus d'asynchronisme.

2.3. Instructions de base

–Affectation : $X := E$. A chaque indice actif u, la valeur $X|_u$ en mémoire privée est mise à jour avec la composante locale de l'expression pure $E|_u$, évaluée en mémoire privée.

–Communication : get X from A into Y. Chaque indice actif u évalue l'adresse $A|_u$ en mémoire privée. La composante locale $Y|_u$ est ensuite mise à jour en mémoire privée avec la valeur distante $X|_{A|_u}$ de la variable X. L'expression $X|_{A|_u}$ est évaluée en mémoire publique de l'indice $A|_u$.

–Copie de mémoires privées : copy. Chaque indice actif copie la composante de chaque variable de sa mémoire privée vers sa mémoire publique.

2.4. Structures de contrôle

–Séquence : $S; T$. Les programmes S et T sont exécutés en séquence. Notons qu'ici la séquence est locale à chaque indice. Elle est donc asynchrone.

–Conditionnelle : where B do S end. Cette construction permet l'exécution concurrente de programmes disjoints. Un indice exécute le bloc d'instructions S si la valeur locale de l'expression pure $B|_u$, évaluée en mémoire privée, vaut *true*. Si la valeur de $B|_u$ s'évalue à *false*, alors la conditionnelle termine. Cette construction n'induit aucune synchronisation.

2.5. Synchronisation

Le langage $\mathcal{SCL}$ autorise les conflits entre les communications et les copies de mémoires induisant des exécutions non déterministes. Nous introduisons à présent une nouvelle instruction sync qui met en œuvre un mécanisme de synchronisation partielle. Cette instruction permet au programmeur de résoudre ces conflits en assurant la cohérence des références aux mémoires publiques.

–Synchronisation partielle : sync. Cette instruction réalise une synchronisation avec l'ensemble des indices *concernés*, c'est-à-dire ceux qui sont actifs pour le bloc d'instructions qui contient l'appel de cette synchronisation.

Les synchronisations ne modifient pas l'interprétation du programme tant que la mémoire publique n'est pas référencée. Nous pouvons par conséquent nous contenter de ne synchroniser un ensemble d'indices actifs que vis-à-vis des instructions qui potentiellement référencent la mémoire publique.

Les instructions get et copy étant les seules à pouvoir y accéder, nous dirons que seules ces instructions sont *significatives* vis-à-vis des synchronisations. La structure de contrôle where détermine quels sont les indices qui vont effectivement modifier la mémoire publique. De plus elle intervient dans l'interprétation d'une synchronisation car elle décide quels sont les indices concernés par celle-ci. La structure where est dite *significative* vis-à-vis des synchronisations. La notion de point de synchronisation est donc relative à la structure et aux instructions significatives.

Discussion : conditionnelle et synchronisations partielles. Les synchronisations sync peuvent ne concerner qu'un sous-ensemble des processeurs virtuels. Ceci est dû à la construction where qui modifie l'ensemble des indices pouvant exécuter l'instruction sync. Par conséquent, une synchronisation partielle termine si et seulement si les deux conditions suivantes sont vérifiées :

C1 : Tous les indices actifs ont évalué les conditions booléennes de tous les where englobants pour décider si ils sont actifs ou non vis-à-vis de cette synchronisation.

C2 : Tous les indices actifs pour cette synchronisation ont atteint le point de synchronisation.

3. Horloges structurelles

Un point crucial est de décider quels indices sont concernés par une synchronisation partielle. Lorsqu'un indice u atteint une synchronisation, il s'informe de la position des autres indices et le point de synchronisation peut être passé uniquement si les conditions C1 et C2 sont remplies. Nous présentons un mécanisme d'horloges structurelles qui modélise la position des indices et permet de gérer les synchronisations partielles. Nous proposons une construction de ces horloges basée sur un étiquetage des instructions du programme et nous définissons un ordre permettant de les comparer.

3.1. Construction des horloges structurelles

Nous associons des étiquettes uniquement aux instructions significatives (copy et get) et à la structure significative (where). Pour un indice donné, la position est définie par le niveau d'imbrication courant, et par l'étiquette de la dernière instruction ou structure significative. La position d'un indice est décrite par une pile qui donne pour chaque niveau d'imbrication sa position dans le bloc. Nous définissons l'étiquette en numérotant les instructions qui accèdent à la mémoire publique à l'intérieur de chaque bloc d'instruction. Le sommet de cette pile donne la position dans le bloc courant. Pour simplifier les notations, les piles seront notées sous forme de listes, où l'élément situé au sommet est placé à droite. Initialement, chaque pile est de hauteur 1, et le sommet contient la valeur 0. Ensuite, les horloges sont modifiées comme suit :

–Quand un indice u exécute un get ou un copy, son horloge est incrémentée de 1 en tête de liste. L'horloge locale $h_u = h.t$ devient $h_u = h.(t+1)$, où le symbole "." désigne l'opérateur de concaténation.

–A l'entrée d'un where, si $B|_u$ est $true$, son horloge $h_u = h.t$ devient $h_u = h.t.0$. Quand l'indice termine le bloc d'instructions associé au where, son horloge $h_u = h.t.s$ devient $h_u = h.(t+1)$. De façon similaire, si $B|_u$ est $false$, son horloge $h_u = h.t$ devient $h_u = h.(t+1)$.

3.2. Ordre sur les horloges structurelles

Nous introduisons un ordre sur les positions, où un indice v est *en retard* sur un indice u si l'indice v ne permet pas de valider l'une des conditions C1 ou C2 pour u. Pour comparer deux horloges structurelles, nous utilisons l'ordre lexicographique. Le terme h_u dénote l'horloge de l'indice u.

Un indice u est dit en retard sur un indice v, et on note $h_u \prec h_v$, si l'une des conditions suivantes est vérifiée :

- *il existe h non vide telle que $h_v = h_u.h$;*
- *il existe h^1, h^2, h^3 telles que $h_u = h^1.t_u.h^2$, $h_v = h^1.t_v.h^3$ et $t_u < t_v$.*

Le premier cas exprime que l'indice v a pris de l'avance sur u : il est entré dans une structure de contrôle que u va rencontrer. Le second cas modélise plusieurs possibilités. Les indices u et v exécutent le même bloc d'instructions à des positions différentes $h^1.t_u$ et $h^1.t_v$. Comme $t_u < t_v$ nous déduisons que l'indice v est en avance sur u dans ce bloc. Si h^2 (resp. h^3) n'est pas vide alors u (resp. v) exécute un bloc d'instructions imbriqué dans celui commun à u et v.

3.3. Réalisation de synchronisations partielles au moyen d'horloges structurelles

L'ordre sur les horloges nous permet de formaliser la notion intuitive de terminaison définie par C1 et C2.

Interprétation : *Considérons un indice u devant exécuter une instruction* sync. *La synchronisation termine s'il n'existe aucun indice v tel que $h_v \prec h_u$.*

Remarque : Un indice peut franchir une synchronisation partielle même si d'autres indices devant participer à cette synchronisation n'ont pas encore atteint le point de synchronisation dans la structure du programme. C'est le cas si toutes les instructions entre les positions courantes de ces autres indices et le point de synchronisation ne sont pas significatives.

L'exemple ci-dessous illustre l'utilisation des horloges structurelles :

```
X := 1;
copy;
where (This < 5) do
   X := 2;
   copy;
end;
where (This > 2) do
   Z := 5 ;
   sync;
   get X from (This − 1) into Y;
end;
...
```

Il n'existe pas d'instruction ni de structure significative entre le second where et le sync. Par conséquent, un indice concerné par la synchronisation peut la franchir dès que tous les indices ont évalué la condition du where : $This > 2$. La synchronisation étant partielle, après évaluation du where, les indices non-concernés par la synchronisation peuvent poursuivre le calcul sans attente.

Le sync garantit le déterminisme de la communication : si un indice exécute le get, alors il a franchi la synchronisation partielle et donc tous les autres indices ont dépassé les copy dans la structure du programme. Après le get, nous avons $Y|_u = 2$ si $3 \leq u \leq 5$ et $Y|_u = 1$ si $u > 5$. Notons que les données lues par le get sont disponibles dès l'exécution de la seconde instruction copy, ce qui rend possible un recouvrement calcul-communication.

4. Conclusion

Nous avons présenté un langage data-parallèle asynchrone augmenté de constructions permettant d'exprimer le synchronisme souhaité. L'interprétation de ce langage repose d'une part sur un modèle d'exécution à mémoires jumelles permettant de masquer certaines transitions locales et d'autre part sur un mécanisme d'horloges structurelles gérant des synchronisations partielles. Contrairement au langage $\mathcal{D}$, l'expression de la concurrence n'est plus bornée par les structures de contrôle.

Cette approche s'étend aux boucles en exprimant la concurrence entre des itérations successives sur des indices distincts [1]. En conséquence, $\mathcal{SCL}$ semble être un candidat intéressant comme langage intermédiaire pour la compilation de langages data-parallèles sur des architectures MIMD. La notion d'horloges structurelles convient tout aussi bien aux architectures à mémoires réparties que partagées, en adaptant le codage de l'algorithme de synchronisations partielles.

Bibliographie

1. Guyadec (Y. L.), Melin (E.), Raffin (B.), Rebeuf (X.) et Virot (B.). – Structural Clocks for a Loosely Synchronized Data-Parallel Language. *In : Proceedings of the 2nd International Conference on Massively Parallel Computing Systems*, éd. par IEEE. – Ishia, Italy, May 1996.
2. Hatcher (P. J.) et Quinn (M. J.). – *Data-Parallel Programming on MIMD Computers*. – The MIT Press, 1991.
3. Le Guyadec (Y.). – Désynchronisation des programmes data-parallèles: une approche sémantique. *TSI*, vol. 14, n° 5, 1995, pp. 619–638.
4. Rice (M. D.). – Semantics for data parallel computation. *International Journal of Parallel Programming*, vol. 19, n° 6, 1990, pp. 477–509.