Parcours d'une superposition de domaines

Olivier Albiez

Ecole des Mines de Paris, Centre de recherche en Informatique
35 rue Saint Honoré, 77305 Fontainebleau, France
e-mail : albiez@cri.ensmp.fr

Résumé
La parallélisation de nids de boucles est l'un des problèmes principaux de la parallélisation automatique. Quand une même transformation est appliquée à toutes les instructions du nid de boucles, le parcours de l'espace d'itérations obtenu est basé sur des techniques bien connues. En revanche, lors de transformations par variables ou par instructions, l'espace d'itérations résultant est une superposition de domaines. Dans cet article, nous proposons un algorithme de parcours d'une superposition quelconque de domaines convexes avec une optimisation du temps d'exécution.

Mots-clés : parallélisation de boucles, domaines paramétrés, transformations affines par instructions

---

Introduction

La parallélisation de nids de boucles est l'un des problèmes principaux de la parallélisation automatique. Pour paralléliser une application, on applique une transformation espace-temps à l'espace d'itérations initial, représenté par des polyèdres convexes. L'espace initial peut être transformé soit par une unique transformation, soit par plusieurs transformations (une par instruction). Quand une même transformation est appliquée à toutes les instructions de la boucle, l'espace d'itérations après transformation est un **Z**-polyèdre[2] et son parcours est basé sur des techniques bien connues[2, 4, 9, 8].

Le problème est plus délicat lorsque différentes transformations sont appliquées, une pour chaque instruction. En effet, le résultat est une superposition de domaines qui n'est plus forcément un **Z**-polyèdre.

Cet article se consacre précisément à cette question : comment parcourir une superposition quelconque de domaines paramétrés convexes. Un domaine paramétré possède des bornes dont les valeurs sont inconnues à la compilation. Il est important de noter que la forme des domaines paramétrés peut être variable en fonction des valeurs de ses paramètres. Dans ce contexte, le code généré doit permettre de tester à l'exécution la valeur des paramètres.

Le code généré peut varier en fonction des critères d'optimisation. Nous avons choisi dans cet article d'optimiser le temps d'exécution au détriment de la taille du code. Nous voulons limiter l'utilisation des tests à l'extérieur des boucles, le prix à payer est donc une duplication du code.

Le parcours de domaines superposés

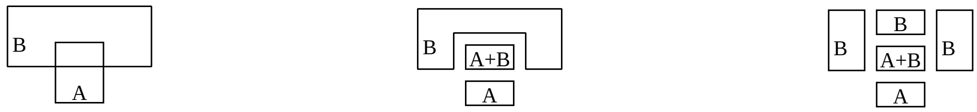
Le parcours est réalisé par des nids de boucles normalisés (les incréments des boucles sont unitaires) et nous nous concentrons uniquement sur la génération des bornes des boucles, pas à la compilation du corps des boucles[1]. Imposer que les boucles soient normalisées signifie que les transformations espace-temps soient définies par des matrices unimodulaires (les domaines sont convexes au départ). Les domaines convexes élémentaires définissant le problème sont donc transformés en polyèdres convexes de $\mathbf{Z}^n$. Une extension aux transformations non unimodulaires qui produit des **Z**-polyèdres est à l'étude.

Posons $\mathcal{E}$ un espace affine de dimension n défini par une base $\mathcal{B} = (b_1, \ldots, b_n)$. Les domaines superposés à parcourir sont définis dans $\mathcal{E}$ et les indices de boucles suivent l'ordre donné par la base $\mathcal{B}$, c'est-à-dire que l'indice le plus externe correspond à b_1 tandis que celui le plus interne correspond à b_n. Le code généré doit respecter les dépendances du nid de boucles initial, donc il doit respecter l'ordre lexicographique défini par $\mathcal{B}$.

L'approche naïve, qui consiste à parcourir l'hypercube contenant tous les domaines et à tester pour chaque point à quel domaine il appartient, requiert beaucoup de tests à chaque itération. En outre il peut y avoir des points de l'hypercube qui n'appartiennent à aucun domaine. On peut limiter le nombre de tests en remplaçant la séquence de tests par une imbrication de tests. Cela correspond à une décomposition des domaines superposés initiaux (Fig. 1.a) en une union de domaines disjoints (Fig. 1.b). Cette opération de décomposition est possible en utilisant les opérations classiques sur les polyèdres. Notons toutefois qu'une union disjointe peut être composée de sous-domaines qui ne sont pas nécessairement convexes comme on peut le voir sur la figure 1.b.

Le code ainsi défini est toujours inefficace en raison du parcours de l'hypercube (il est souvent plus grand que l'union des domaines à parcourir) et en raison des tests restants. On peut résoudre ces deux problèmes en remplaçant le nid de boucles parfait par une séquence de nids de boucles où chaque boucle interne décrit un sous-domaine **convexe** de l'union disjointe. En effet, dans ce cas, seuls les points nécessaires sont parcourus et chaque nid de boucles ne décrit qu'un seul sous-domaine (les tests deviennent inutiles).

La construction de la séquence de nids de boucles est réalisée par un découpage de l'union disjointe pour obtenir une **union disjointe de domaines convexes** (Fig. 1.c).

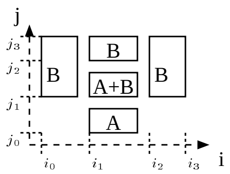


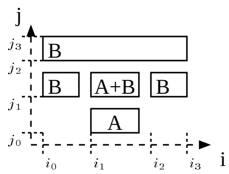
(a) Domaines superposés non disjoints — (b) Union disjointe de domaines non nécessairement convexes — (c) Union disjointe de domaines convexes

FIG. 1 - *Décomposition de domaines superposés*

Ce découpage est dirigé en fonction de l'ordre de parcours et doit être réalisé avec précaution pour éviter d'inutiles accroissements de code. La figure 2 présente deux découpages différents de la figure 1.b. Le découpage (a) est approprié pour un parcours en i puis j et le nid de boucles s'en déduit simplement. En revanche, (b) est inapproprié pour un parcours en i puis j car on ne peut pas en déduire un seul nid de boucles, il faut le redécouper.

```
DO i = i0, i1 - 1
    DO j = j1, j3
        B
    END
END
DO i = i1, i2 - 1
    DO j = j0, j1 - 1
        A
    END
    DO j = j1, j2 - 1
        A + B
    END
    DO j = j2, j3
        B
    END
END
DO i = i2, i3
    DO j = j1, j3
        B
    END
END
```





Pas de nids de boucles permettant de parcourir en i puis j ces polyèdres sans les redécouper

(a) Découpage approprié — (b) Découpage inapproprié

FIG. 2 - *Découpage des domaines non convexes pour un parcours en i puis j*

Avec un tel découpage, nous avons supprimé les tests sur les indices, mais les tests internes cachaient des tests sur les paramètres. Notre objectif suivant est de déplacer ces tests à l'extérieur de la séquence de nids de boucles et de traiter ainsi les paramètres **avant** tous les indices. Ces tests sont basés sur les domaines de validité des paramètres. Pour chaque domaine de validité, le code à exécuter est spécifique, la forme des domaines à parcourir pouvant être différente. Il y aura en conséquence des duplications de codes.

Polyèdres paramétrés : définitions et opérations

Les objets manipulés dans cet article sont des polyèdres convexes paramétrés. Par la suite, nous noterons $p \in \mathbf{Z}^m$ le vecteur des paramètres. Un polyèdre paramétré peut être défini par sa **représentation**

implicite en utilisant un ensemble de contraintes linéaires :

$$\mathcal{D}_p = \{x \in \mathbf{Z}^n \mid Ax = Bp + b, Cx \geq Dp + d\}$$

avec A une matrice entière de taille $j \times n$, B une matrice entière de taille $j \times m$, b un vecteur entier de taille j, j est le nombre d'égalités, C une matrice entière de taille $k \times n$, D une matrice entière de taille $k \times m$, d un vecteur entier de taille k, k est le nombre d'inégalités.

Un tel polyèdre peut être réécrit comme un polyèdre non paramétré dans l'espace combiné données/paramètres de dimension $n + m$[6]. Nous l'appelons le **polyèdre combiné**:

$$\mathcal{D}' = \left\{ \begin{pmatrix} x \\ p \end{pmatrix} \in \mathbf{Z}^{n+m} \mid A' \begin{pmatrix} x \\ p \end{pmatrix} = b,\ C' \begin{pmatrix} x \\ p \end{pmatrix} \geq d \right\}$$

avec $A' = [A| - B]$ et $C' = [C| - D]$.

Comme tout polyèdre non paramétré, un polyèdre combiné peut être défini par sa **représentation duale de Minkowski**. Il est alors exprimé comme une combinaison de lignes, rayons et sommets. Ces deux représentations, implicite et Minkovski, sont équivalentes et il existe des algorithmes pour passer de l'une à l'autre.

Outre les opérations classiques sur les polyèdres comme l'union disjointe ou l'intersection[11], nous utilisons les deux opérations suivantes de la Polylib :

- la **simplification**. Notée $C = \texttt{Simplify}(A, B)$, cette opération supprime de la représentation implicite de A toutes les contraintes de B qui sont redondantes avec celles de A. Cela revient à trouver le polyèdre C défini par

  $$A \cap B = C \cap B \ \ (\text{et} \ \ \nexists C' \text{ tel que } C \subset C',\ A \cap B = C' \cap B)$$

  Dans *Omega*[3], cette opération est apellée `Gist`.

- l'**extension**. Notée $C = \texttt{AddRays}(A, R)$ avec A, C des polyèdres et R un ensemble de lignes, rayons et sommets. Cette opération construit l'enveloppe convexe de A augmentée des éléments de R.

L'algorithme de génération de code

L'algorithme que nous avons développé analyse le domaine d'itérations défini dans l'espace combiné. Comme nous souhaitons que les tests sur les paramètres soient effectués à l'extérieur des boucles, nous considérons les paramètres comme de nouveaux indices que nous parcourons en premier. L'ordre des indices de boucles est défini par une liste ordonnée (du plus externe au plus interne, paramètres inclus). La première étape fragmente les domaines superposés en une union disjointe de domaines non nécessairement convexes. Ensuite, on appelle la fonction principale (`Loop-Generator`) de l'algorithme qui est récursive sur le niveau de profondeur des boucles. Elle utilise un contexte qui définit le polyèdre engendré par les boucles externes et fonctionne de la manière suivante.

1. La première opération effectuée est le découpage des domaines en entrée en fonction de l'ordre des indices (figure 2). Ce découpage est réalisé par la fonction `Split` qui est détaillée plus loin.

2. Le domaine découpé est ensuite projeté selon les indices internes (ceux qui restent à parcourir). Le résultat est un ensemble de polyèdres où toutes les références aux indices internes sont supprimées. L'indice courant est donc exprimé en fonction des indices externes (utilisation de la fonction `AddRays` de la PolyLib).

3. Le polyèdre résultant est alors simplifié en supprimant toutes les contraintes redondantes relatives aux indices externes (utilisation de la fonction `Simplify` de la PolyLib).

4. Chacun de ces polyèdres contient les contraintes linéaires caractérisant les bornes supérieures et inférieures de l'indice courant. Les boucles correspondantes sont alors générées.

5. Pour chaque boucle précédemment générée, on ajoute ses bornes au contexte courant afin de générer les boucles internes (appel récursif de `Loop-Generator`).

Le découpage des domaines non convexes de l'union disjointe est réalisé par la fonction `Split`. L'algorithme de découpage calcule pour chaque domaine l'ensemble des contraintes relatives à l'indice courant en fonction des indices externes. Ces contraintes servent à partitionner les domaines d'entrée.

Rappelons que seuls les polyèdres convexes sont définis par un ensemble de contraintes. Comme les domaines d'entrée de la fonction `Split` viennent d'une union disjointe, ils peuvent être non convexes. Cependant, cela ne pose pas de problème car de tels domaines non convexes sont représentés en interne comme une union de polyèdres convexes et la fonction examine dans ce cas tous les polyèdres élémentaires pour calculer les contraintes.

Conclusion

La génération de code dans le contexte de transformations multiples a été traitée par peu d'auteurs. Quinton, Rajopadhye et Wilde ont montré comment produire un code parallèle en partant d'un code *Alpha*[7]. Leur approche est également basée sur la manipulation de polyèdres. La principale différence est qu'ils ne traitent pas les paramètres formels, en effet ceux-ci doivent être instanciés avant la génération de code. Kelly, Pugh et Rosser ont développé un générateur de code dans *Omega*[3]. Il est basé sur l'utilisation des formules de Presburger et d'arbres syntaxiques abstraits qui représentent la structure des boucles. De cette représentation sous forme d'arbres, le code est généré suivant différentes optimisations (l'algorithme prend en entrée une information pour guider l'optimisation).

De notre coté, nous travaillons avec les transformations géométriques de polyèdres convexes en utilisant la librairie *Polylib*[10]. Le code généré est optimisé pour minimiser le temps d'exécution. L'algorithme a été intégré dans *OPERA*, un environnement pour la parallélisation de systèmes d'équations récurrentes affines[5]. Notre algorithme devrait être amélioré pour proposer différentes optimisations, en outre, une étude complète devrait être conduite pour traiter des transformations multiples non unimodulaires. De telles transformations soulèvent la question du parcours de **Z**-polyèdres.

Bibliographie

1. Albiez (O.). – *Génération de code dans le cadre des transformations affines par variable.* – Rapport de DEA, ICPS, Strasbourg, France, 1997.
2. Ancourt (C.) et Irigoin (F.). – Scanning polyhedra with DO loops. *In : Proc. ACM SIGPLAN '91*, pp. 39–50.
3. Kelly (W.), Pugh (W.) et Rosser (E.). – Code generation for multiple mappings. *In : 5th Symposium on Frontiers of Massively Parallel Computation*, pp. 332–341. – McLean, Virginia.
4. Li (W.) et Pingali (K.). – A singular loop transformation framework based on non-singular matrices. *In : Proc. of the 5th Workshop on Languages and Compilers for Parallel Computing.* – Yale University.
5. Loechner (V.) et Mongenet (C.). – Opera : a toolbox for loop parallelization. *In : First IFIP Worshop on software engineering for parallel and distributed systems, Berlin Germany*, éd. par Jelly (I.), Gorton (I.) et Croll (P.). pp. 134–145. – Chapman and Hall.
6. Loechner (V.) et Wilde (D. K.). – Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, vol. 25, n° 6, December 1997.
7. Quinton (P.), Rajopadhye (S.) et Wilde (D.). – Deriving imperative code from functional programs. *In : FPCA'95.* – La Jolla, California.
8. Quinton (P.), Rajopadhye (S. V.) et Risset (T.). – On manipulating Z-polyhedra using a canonical representation. *Parallel Processing Letter*, vol. 7, n° 2, 1997, pp. 181–194.
9. Ramanujam (J.). – Non-unimodular transformations of nested loops. *In : Supercomputing 92.* pp. 214–223. – IEEE Computer Society Press.
10. Wilde (D.). – *A library for Doing Polyhedral Operations.* – Rapport technique n° Internal Publication 785, Rennes, France, IRISA, Dec 1993.
11. Wilde (D.). – *From Alpha to Imperative Code: A Transformational Compiler for an Array Based Functional Language.* – Thèse de PhD, Oregon State University, 1995.