Polyèdres et Compilation

François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Ronan Keryell

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Polyèdres et Compilation

François IRIGOIN[1] & Mehdi AMINI[1,2] & Corinne ANCOURT[1] &
Fabien COELHO[1] & Béatrice CREUSILLET[2,3] & Ronan KERYELL[2]

[1]MINES ParisTech, Maths & Systèmes, CRI, Fontainebleau, France
[2]HPC Project, Meudon, France
[3]UPMC/LIP6, Paris, France

Résumé
La première utilisation de polyèdres pour résoudre un problème de compilation, la parallélisation automatique de boucles en présence d'appels de procédure, a été décrite et implémenté il y a près de trente ans. Le modèle polyédrique est maintenant reconnu internationalement et est en phase d'intégration dans le compilateur GCC, bien que la complexité exponentielle des algorithmes associés ait été pendant très longtemps un motif justifiant leur refus pur et simple. L'objectif de cet article est de donner de nombreux exemples d'utilisation des polyèdres dans un compilateur optimiseur et de montrer qu'ils permettent de poser des conditions simples pour garantir la légalité de transformations.

Mots-clés : transformation de programme, synthèse de code, parallélisation, polyèdre, interprétation abstraite

---

1. Problèmes et théories rencontrés en compilation

Le développement des compilateurs depuis les années 50 a conduit à des développements théoriques maintenant bien maîtrisés ainsi qu'au découpage de ces outils en trois grandes parties appelées en anglais *front-end*, *middle-end* et *back-end*. La première partie est censé traiter les spécificités du langage d'entrée, la troisième celles de la machine cible tandis que le *middle-end* est censé être indépendant à la fois du langage d'entrée et de la machine cible grâce à une représentation interne bien choisie.
Les développements théoriques suivants sont généralement reconnus comme liés à la compilation, ainsi que le montre la table des matières du *Dragon Book* [2, 1] :

Front-end Analyses lexicale et syntaxique $\Longrightarrow$ théorie des automates

Middle-end 1) analyses de graphes de contrôle ou de données(CFG, def-use chains, DDG, PDG, SSA,...), composantes connexes ou fortement connexes, 2) ordonnancement, placement, réallocation mémoire, 3) analyses statiques : équations de flots de données, i.e. équations aux ensembles, finis ou infinis

Back-end Pattern-matching, coloration de graphes, techniques d'ordonnancement.

Par ailleurs, l'analyse et la vérification de programmes ont conduit d'une manière largement indépendante à l'interprétation abstraite et au model checking. De nombreux domaines abstraits ont été utilisés afin de pouvoir faire un compromis entre la précision des informations obtenues et la complexité des opérateurs. Un domaine abstrait particulier, introduit par Halbwachs dans sa thèse [19], celui des polyèdres, a particulièrement retenu notre attention au début des années 80 parce qu'il permettait de représenter exactement des ensembles d'éléments de tableaux et des ensembles d'itérations et donc d'attaquer le problème de la parallélisation interprocédurale.
Dans le *middle-end* d'un compilateur classique, les équations de flots de données sont des équations aux ensembles, mais les ensembles utilisés sont souvent finis et représentés par des vecteurs de bits. Mais pourquoi se limiter à des vecteurs de bits dans un compilateur quand on peut profiter des diverses surapproximations d'ensembles infinies développées dans le cadre de l'interprétation abstraite ?
La raison essentielle apportée par les américains, à commencer par Ken Kennedy, est qu'il est impensable d'utiliser un algorithme de complexité exponentielle dans un compilateur. L'objet essentiel de ce

papier est de faire le recensement des utilisations possibles de la théorie des polyèdres[1] dans le *middle-end* d'un compilateur et donc d'en montrer l'intérêt pour le concepteur.
La question de la maîtrise de la complexité a conduit à l'introduction de très nombreuses abstractions finies d'ensembles infinis, que ce soit pour représenter des valeurs, des cases mémoire, des itérations ou bien des dépendances entre itérations : signe, parallélépipèdes (produit d'intervalles), octogones, polyèdres, Z-polyèdres, listes finies de polyèdres, arithmétique de Presburger, fonctions affines par morceaux (QUAST),...
Nous sommes particulièrement attachés à l'abstraction polyédrique parce qu'elle a toujours donné de très bons résultats expérimentaux avec notre outil PIPS [34] et parce que nous pensons que la complexité exponentielle en magnitude, espace et temps des opérateurs associés peut être maîtrisée, mais nous essayons de faire une présentation aussi générique que possible en nous attachant aux objets mathématiques sous-jacents que sont 1) les ensembles, 2) les produits cartésiens d'ensembles, 3) les relations et graphes de relations, et 4) les fonctions et graphes de fonctions, et les applications.
Nous présentons donc un ensemble de composants expérimentaux faisant appel à des représentations abstraites de ces objets. Ils ont été développés dans le cadre de l'outil logiciel PIPS (parallélisation interprocédurale de programme scientifique) au fil des ans par Rémi Triolet [50], François Irigoin [32, 31], Corinne Ancourt [4, 8], Yi-Qing Yang [54, 55], Ronan Keryell [5], Fabien Coelho [15], Béatrice Creusillet [22], Serge Guelton [29], Mehdi Amini [3], et bien d'autres chercheurs à commencer par Pierre Jouvelot.
La présentation est organisée en fonction des structures utilisées, ensembles, relations et fonctions. L'abstraction choisie est presque toujours l'abstraction polyédrique dans PIPS. Elle consiste à représenter un ensemble d'entiers par des polyèdres rationnels le contenant et constituant une sur-approximation en général.

2. Utilisations d'ensembles

La notion d'ensemble est à la fois simple, bien connue et puissante. Les prédicats et opérateurs sur les ensembles sont bien définis et il existe des algorithmes de complexité diverses pour les implanter dans l'abstraction polyédrique en utilisant l'une ou l'autre des deux représentations finies, le système de contraintes affines ou le système générateur avec sommets, rayons et droites [48].
Comme toute abstraction, l'abstraction polyédrique conduit à considérer des sur- ou des sous-approximations, voir des ensembles exacts dans certains cas. Des algorithmes particuliers doivent être utilisés pour maintenir des informations sur la nature des approximations : inférieures, supérieures ou bien encore égales.
Quatre types d'ensembles au moins sont utilisés dans un compilateur : les ensembles de valeurs, les ensembles d'itérations, les ensembles d'éléments de tableaux et les ensembles de dépendances.

2.1. Ensembles de valeurs

Les ensembles de valeurs sont utilisés pour effectuer la propagation de constantes, la simplification d'expressions, la simplification du contrôle et la substitution des variables d'induction. Les ensembles de valeurs, ou états mémoire, sont notés P, comme précondition, quand il s'agit de l'état mémoire existant juste avant l'exécution d'une instruction. Chaque état mémoire est noté σ, comme *store*.

2.1.1. Préconditions sur les scalaires

L'ensemble des valeurs prises par les variables scalaires du programme avant l'exécution d'une instruction est appelé précondition de cette instruction. Cet ensemble de valeurs n'est généralement pas connu exactement, mais la théorie de l'interprétation abstraite [17] permet d'en calculer des sur-approximations en utilisant divers domaines abstraits qui sont souvent des sous-ensembles de l'ensemble des polyèdres convexes. Les domaines abstraits les plus simples sont tout simplement les signes des variables ou leurs intervalles de variation ou bien le domaine des constantes. Le domaine des octogones [41] a aussi été proposé afin de réduire la complexité des opérateurs nécessaires.

[1] Le modèle polyédrique, maintenant bien connu[27], est, au sens strict, un cas particulier d'utilisation des polyèdres pour traiter des problèmes d'ordonnancement, de placement et d'allocation pour les parties de programme à contrôle statique [37]. Au sens large, n'importe quel algorithme utilisant un polyèdre pour faire de la compilation relève du modèle polyédrique.

2.1.2. Propagation de constantes et simplification d'expressions

La propagation de constantes, à savoir la substitution d'une référence à une variable scalaire par sa valeur, est plus connue que la simplification d'expressions qui consiste à trouver une nouvelle expression équivalente par rapport aux états mémoire possible lors de son évaluation.
Les états mémoire σ sont restreints par un prédicat P appelé précondition. Soit $\mathcal{E}()$ la fonction permettant d'évaluer une expression e et $S(e)$ l'ensemble des expressions équivalentes à e :

$$S(e) = \{e' \in \text{EXPR} | \forall \sigma \in P \ \mathcal{E}(e, \sigma) = \mathcal{E}(e', \sigma)\}$$

Une expression est considérée comme simplifiée si le nombre de références et/ou d'opérations nécessaires à son évaluation est réduit. Les expressions étant définies récursivement, on peut essayer de simplifier les sous-expressions jusqu'à atteindre les références et les opérations élémentaires, mais on peut aussi essayer de plonger l'expression dans l'abstraction utilisée pour les valeurs en utilisant de nouvelles dimensions.
Il faut associer à chaque opérateur concret un opérateur abstrait et écrire les équations et contraintes correspondantes. Par exemple, avec l'ensemble des polyèdres, le cas de l'addition est simple :

$$\mathcal{E}(e1 + e2) = \mathcal{E}(e1) + \mathcal{E}(e2)$$

tout comme la soustraction. Par contre, les opérateurs multiplication, division, modulo et décalage nécessitent l'ajout de nombreuses contraintes qui dépendent de ce qui peut être observé pour chaque sous-expression. Cette fonctionalité est implémentée de manière générique dans l'interface de niveau 1 d'APRON [35].
On finit néanmoins par atteindre les références à des variables comme n. Il suffit d'éliminer les quantificateurs liés à σ en projetant la précondition sur la dimension correspondant à n. En notant Σ_P la fonction qui retourne l'ensemble des valeurs correspondant à un identificateur pour une précondition P, on obtient :

$$\Sigma_P(n) = \{v \in \text{Val} | \exists \sigma \in P \ \sigma(n) = v\}$$

La référence à n peut être surrpimée si cet ensemble est un singleton.
Opérateur : la projection, comme élimination de quantificateur.

2.1.3. Simplification du contrôle

La simplification du contrôle consiste à éliminer les tests et leurs branches inutiles, ainsi que les boucles ayant zéro ou une itération. Dans certains cas, on peut aussi éliminer le code qui suit une boucle infinie ou certains appels comme exit(), abort(),... : il suffit que sa précondition représente l'ensemble vide. Il n'existe alors aucun état mémoire possible avant l'exécution de ce code.
Soit c la condition dont on veut montrer qu'elle est toujours vraie sous la précondition P. La condition s'écrit en montrant qu'il n'existe aucun état dans lequel elle est fausse :

$$\{\sigma | P(\sigma) \wedge \mathcal{E}(\bar{c}, \sigma)\} = \emptyset$$

On procède de la même manière pour montrer qu'une condition est toujours fausse.
P peut être utilisée de manière similaire pour vérifier qu'une boucle `DO`, `for` ou `while` n'est jamais exécutée. Enfin, dans le cas contraire, l'expression donnant le nombre d'itérations peut être évaluée, comme n'importe quelle autre expression, pour obtenir le nombre d'itérations exactement, quand c'est possible, ou bien au moins un intervalle ce qui peut être utile pour estimer si une boucle peut être exécutée en parallèle de manière profitable.
Opérateurs : projection, test à vide

2.1.4. Substitution des variables d'induction

Les variables d'induction peuvent être détectées dans les boucles en faisant du pattern-matching sur leurs initialisations et mises à jour, mais, si l'on dispose de préconditions relationnelles, il suffit de vérifier qu'il existe dans la précondition du corps de boucle une équation entre la variable d'induction supposée et l'indice de la boucle courante. Pour ce faire, il suffit de projeter toutes les autres valeurs. Ce

test peut être fait avant chaque référence à une variable dans le cas où celle-ci est mise à jour plusieurs fois dans le corps de boucle. Ceci devient une simple extension de la simplification d'expression avec le remplacement d'une expression par une expression équivalente mais non nécessairement plus simple.
La variable k peut être remplacée par une expression dans l'instruction S de précondition P_S dans une boucle d'indice i si P_S définit une application de $\sigma(i)$ vers $\sigma(k)$:

$$\nu \rightarrow \{\nu' | \exists \sigma \in P_S \ \sigma(i) = \nu \wedge \sigma(k) = \nu'\}$$

En d'autres mots, on vérifie que la précondition est suffisament précise pour associer à chaque itération une valeur unique pour k.
Au niveau du corps de boucle, on peut aussi déterminer pour quelles variables il existe une équation de transition et en déduire par intégration quelles sont parmi elles les variables d'induction.
Ceci donne une condition suffisante de légalité, mais n'indique ni si la transformation est utile, ni comment la faire. De plus la qualité des résultats dépend de la précision de P_S, mais la légalité est toujours garantie par une sur-approximation.

2.2. Ensemble d'itérations

La structure de boucle conduit naturellement à définir des ensembles d'itérations monodimensionnels. L'imbrication de boucles, voire la position textuelle des instructions se trouvant dans le corps des boucles, conduisent à augmenter le nombre de dimensions et à utiliser l'algèbre linéaire pour exprimer des transformations de programme dans la mesure où les bornes de boucles sont affines ou transformables en une expression affine.
En calcul scientifique, les bornes de boucles définissent très fréquemment des ensemble d'itérations polyédriques, fonctions de paramètres. On dispose donc de l'algèbre linéaire pour représenter les transformations de boucles, i.e. les réordonnancements, et des polyèdres pour régénérer les boucles transformées.

2.2.1. Transformations unimodulaires de boucles

Une transformation unimodulaire, échange de boucles ou méthode hyperplane, consiste à effectuer un changement de base sur l'espace affine dans lequel se trouve l'ensemble des itérations. On veut donc passer de l'ensemble d'itérations $I = \{i | Bi \leq b\}$, ou B et b sont dérivés des bornes de boucles, à l'ensemble d'itérations $J = \{j | \exists i \ Bi \leq b \wedge j = Mi\}$ où M est la matrice de transformation unimodulaire. Comme M est inversible et qu'elle transforme des points entiers en points entiers, on obtient le nouvel ensemble d'itérations $J = \left\{j | BM^{-1}j \leq b\right\}$. Il faut ensuite régénérer les boucles à partir de ce nouveau polyèdre.
Opérateurs : produit de matrice, inversion de matrice (élimination de quantificateurs), énumération des points entiers d'un polyèdre.

2.2.2. Tiling

Le tiling [31] est une transformation non-unimodulaire ce qui ne permet pas de passer par une simple inversion de matrice pour éliminer les quantificateurs. De plus il faut calculer les boucles sur les tuiles ainsi que les boucles dans les tuiles.
Soit T l'espace des tuiles, S, L, s le système définissant les tuiles, leur treillis et leur origine, et t les coordonnées d'une tuile. Soit I l'ensemble des itérations, (B, b) les bornes de boucles initiales et i une itération quelconque. On obtient alors l'ensemble exact d'itérations des tuiles t :

$$\{t \in T | \exists i \in I \text{ t.q. } Bi \leq b \wedge S(i - Lt) \leq s\}$$

Pour obtenir une sur-approximation de cet ensemble, il faut éliminer la quantification sur i, par exemple en effectuant une projection de i. On obtient alors les bornes sur t sous la forme de contraintes affines, (B_T, b_T).
Opérateurs : produit de matrices, projection par Fourier-Motzkin[2]

[2] C'est Paul Feautrier qui a suggéré à Rémi Triolet l'utilisation de l'élimination dite de Fourier-Motzkin pour réaliser un test de dépendance.

2.2.3. Ordonnancement par fonctions affines

L'ordonnancement par fonctions affines a été proposé par Paul Feautrier en 1991/92 [25, 26] et a lancé le *modèle polyédrique* en compilation. Son histoire a été présentée lors de précédentes rencontres du parallélisme en 2002 [27]. Les extensions récentes de ces travaux utilisent toujours une représentation polyédrique de l'ensemble des occurrences des instruction. Mais la particularité de cette école est de n'utiliser que des représentations exactes de cet ensemble d'occurrences et de l'ensemble des dépendances existant entre elles.

Ces ordonnancements correspondent à des transformations de programme beaucoup plus complexes que de simples transformations de nids de boucle, mais Cédric Bastoul a développé un algorithme permettant de régénérer un contrôle de bonne qualité [10].

Opérateurs : PIP, CLooG,...

2.3. Ensemble d'éléments de tableaux

Les ensembles d'éléments de tableaux sont utilisés pour 1) optimiser et générer les communications explicites quand plusieurs mémoires existent, qu'il s'agisse d'une machine à mémoire répartie ou d'une machine munie d'un accélérateur, GPU ou FPGA (sections 2.3.1 à 2.3.5), 2) adapter les paramètres de *tiling* à une hiérarchie mémoire (section 2.3.6) et 3) vérifier que les accès aux tableaux sont conformes aux déclarations ou bien réciproquement pour adapter les déclarations aux utilisations des tableaux (section 2.3.7).

On peut avoir besoin de connaître exactement un ensemble d'éléments à écrire pour ne pas perturber un calcul, ce qui fait sortir du cadre habituel de l'interprétation abstraite et nécessite de nouveaux algorithmes [8, 46].

2.3.1. Génération des communications

La recherche de puissances de calcul élevées ou d'une grande efficacité énergétique passe par la réalisation de systèmes dont la mémoire est répartie de manière explicite pour le programmeur. L'exécution d'un programme comprend alors presque nécessairement des phases de synchronisation et d'échanges de données.

De nombreuses tentatives ont été effectuées et le sont encore pour éviter au programmeur d'avoir à spécifier exactement quelles étaient les données à transférer et pour le laisser les spécifier implicitement.

Le développement de la norme et des compilateurs HPF, *High-Performance Fortran*, n'a pas abouti à des résultats satisfaisants mais elle a permis de faire progresser la génération de communications, d'allocations mémoire et du contrôle [7] en utilisant l'algèbre linéaire (section 2.3.2). Nous avons aussi fait un essai plus limité pour gérer par logiciel des bancs mémoire (section 2.3.3). L'arrivée des processeurs *manycore* devrait renouveler l'intérêt pour ces techniques.

Plus récemment, une équipe de Telecom SudParis a entrepris avec succès de traduire automatiquement du code OpenMP en code MPI, toujours en caractérisant manière polyédrique les ensembles de données à transférer (section 2.3.4).

Enfin, l'arrivée des GPU et la difficulté qu'elles posent pour partager efficacement et de manière cohérente leur bus mémoire avec la ou les CPUs associées conduit à nouveau à étudier le placement de données sur des mémoires réparties, ainsi que la minimisation et la génération automatique des transferts induits par la répartition (section 2.3.5).

Il n'est pas possible dans le cadre d'un article de présenter toutes ces techniques en détail. Nous nous limitons à quelques exemples et à quelques références bibliographiques à l'intention des lecteurs les plus intéressés.

2.3.2. Compilation HPF

Différentes approches ont couramment été proposées pour utiliser un environnement de mémoire partagée sur une architecture à mémoire distribuée. Le langage HPF a été développé pour exécuter des programmes SPMD sur des machines à mémoire répartie. Le programmeur précise la distribution des données sur les processeurs par le biais de directives. Le compilateur exploite ces directives pour allouer les tableaux en mémoire locale, affecter les calculs propres à chaque processeur élémentaire et générer les communications entre les processeurs.

Dans le cadre d'HPF, la règle *owner compute rule*[3] est utilisée et les processeurs exécutent les instructions si les mises à jour des données correspondent à des données locales. Les communications sont déduites de la distribution des données.
Un cadre algébrique affine a été proposé [7] pour modéliser et compiler HPF. La distribution des données est précisée en deux temps. Les éléments de tableaux sont *alignés* sur des processeurs virtuels appelés des *templates*. Ensuite les *templates* sont *distribués* sur les processeurs par blocs ou de manière cyclique. Les contraintes modélisant la distribution et l'*alignement* des données sur les processeurs sont résumées dans ce système :

$$\left\{0 \leq D^{-1}\vec{a} < 1\ , 0 \leq T^{-1}\vec{t} < 1\ , 0 \leq P^{-1}\vec{p} < 1\ , 0 \leq C^{-1}\vec{l} < 1\ , \wp\vec{t} = A\vec{a} + \vec{t}_0\ , \Pi\vec{t} = CP\vec{c} + C\vec{p} + \vec{l}\right\}$$

Les premières contraintes reprennent respectivement les déclarations des tableaux, des *templates*, des processeurs et des tailles de blocs précisées par la directive de distribution.
La cinquième contrainte donne l'alignement des éléments $\vec{a}$ du tableau sur un *template* $\vec{t}$. Il peut s'exprimer dimension par dimension comme combinaison linéaire d'éléments des *templates*. Afin de pouvoir traduire la réplication éventuelle d'éléments du tableau sur des dimensions différentes du *template*, une matrice de projection $\wp$ supplémentaire est nécessaire.
La distribution des *templates* par bloc sur les processeurs se traduit comme une relation (6-ième équation) entre les coordonnées des processeurs $\vec{p}$, celles du *template* $\vec{t}$ et deux autres variables : $\vec{l}$ et $\vec{c}$. $\vec{l}$ représente l'offset dans un bloc sur un processeur et $\vec{c}$ le nombre de cycles nécessaires pour allouer les blocs sur les processeurs. La matrice Π est utile lorsque plusieurs dimensions du *template* sont rassemblées sur un même processeur. P est une matrice diagonale et décrit la géométrie de la machine. Chaque élément de sa diagonale est égale au nombre de processeurs pour chaque dimension.
Afin de générer le code correspondant aux éléments référencés par le noyau de calcul, il faut ajouter l'ensemble des contraintes reliant le domaine d'itérations des nids de boucles de calcul et les références au tableau :

$$B\vec{\imath} \leq \vec{b}, \vec{a} = R\vec{\imath} + \vec{r}$$

Maintenant nous pouvons définir l'ensemble $Own_X(p)$ des éléments d'un tableau X placés sur un processeur $\vec{p}$.

$$Own_X(\vec{p}) = \{\ \vec{a} \mid \exists\vec{t}\ ,\ \exists\vec{c}\ ,\ \exists\vec{l}\ \text{t.q.}\quad 0 \leq D_X^{-1}\vec{a} < 1\ ,\ 0 \leq T_X^{-1}\vec{t} < 1\ ,\ 0 \leq P^{-1}\vec{p} < 1\ ,\ 0 \leq C_X^{-1}\vec{l} < 1\ ,\ \wp\vec{t} = A\vec{a} + \vec{t}_0\ ,\ \Pi\vec{t} = C_X P\vec{c} + C_X\vec{p} + \vec{l}\ \}$$

En utilisant la règle *owner compute*, pour un noyau d'instructions calculant les valeurs d'un tableau X, nous dérivons l'ensemble des itérations devant être exécutées sur le processeur $\vec{p}$:

$$Compute(\vec{p}) = \{\vec{\imath} \mid R_X\vec{\imath} + \vec{r}_X \in Own_X(\vec{p})\ \wedge\ B\vec{\imath} \leq \vec{b}\}$$

Ainsi l'ensemble des éléments d'un tableau Y lus par ce même noyau est défini par :

$$View(\vec{p}) = \{\vec{a} \mid \exists\vec{\imath} \in Compute(\vec{p})\ \text{t.q.}\ \vec{a} = R_Y\vec{\imath} + \vec{r}_Y\}$$

Plusieurs solutions sont proposées dans [7] pour optimiser l'allocation des tableaux locaux à un processeur et les *Views*, notament lorsque les calculs référencent plusieurs fois le même tableau. Les ensembles d'éléments qui appartiennent à un processeur p et qui sont utilisés par un autre processeur p' doivent être transférés. Ils sont définis par les ensembles suivants :

[3] Cette règle stipule qu'un calcul est effectué sur la machine où est stocké le résultat.

$$Send_Y(p, p') = Own_Y(p) \bigcap View_Y(p')$$
$$Receive_Y(p, p') = View_Y(p) \bigcap Own_Y(p')$$

$Send$ et $Receive$ représentent des ensembles non nécessairement convexes. Il faut rester vigilant lors de la génération de code afin d'obtenir un code efficace tenant compte : de l'allocation des tableaux, de leurs *layouts*, évitant les réplications ou communications inutiles d'un processeur vers lui même, n'introduisant pas de tests superflus, ..., plus de détails sont présentés dans [7]. D'autres algorithmes tels que [4, 8, 10] ont également été développés.
Opérateurs : projection, élimination de contraintes redondantes, génération de bornes de boucles

2.3.3. Émulation logicielle de bancs mémoire : WP65

Dans le cadre du projet européen PUMA et du workpackage WP65 [4, 9], une mémoire partagée virtuelle fournissait un espace d'adressage uniforme.
Une partie des processeurs de l'architecture distribuée émulent les bancs de la mémoire virtuelle partagée[4]. Les données sont uniformément distribuées sur ces bancs mémoire. Les autres processeurs constituent les unités de calcul. Les calculs ne dépendent que du contrôle du programme. Les tâches parallèles sont calculées en fonction des dépendances entre les instructions du programme. La distribution des données est donc indépendante de leur utilisation.
Dans le cadre du projet, des transformations de boucles telles que le tiling ont été appliquées pour définir les blocs d'instructions pouvant être exécutées en parallèle et ajuster leur taille de manière à ce que les temps de communication ne soient pas supérieurs aux temps de calcul.
Chaque tâche est constituée de 3 parties : un prologue qui lit les données nécessaires aux calculs (communication des bancs mémoire vers les processeurs), une partie correspondant aux calculs, et un épilogue correspondant à l'écriture des résultats (communication des processeurs vers les bancs mémoire).
Chaque processeur de calcul exécute ses instructions en ne référençant que les données transférées dans sa mémoire locale. L'ensemble des éléments de tableaux lus lors de l'exécution doit donc être copié en mémoire locale. Cet ensemble peut être approximé et surestimé sans que cela n'introduise de problème de cohérence mémoire puisqu'il s'agit d'éléments lus.
Une fois les calculs terminés, les données modifiées doivent être recopiées dans la mémoire partagée. Ces ensembles d'éléments ne sont ni nécessairement convexes, et ne correspondent pas systématiquement à des éléments contigus sur les bancs mémoire. Pour ne pas introduire de conflits et respecter les dépendances, seuls les éléments modifiés doivent être transférés.
L'ensemble des éléments $\vec{a}$ du tableau A est représenté par l'ensemble suivant :

$$\{\vec{a} \in A \mid \exists \vec{\imath}' \in I' \text{ t.q. } C(\vec{\imath}') \wedge \vec{a} = R\vec{\imath}' + \vec{r}\} =$$
$$\{\vec{a} \in A \mid \exists \vec{\imath} \in I \text{ t.q. } B\vec{\imath} \leq \vec{b} \quad \wedge (\vec{a} = R_1(\vec{\imath}) + \vec{r}_1 \vee \vec{a} = R_2(\vec{\imath}) + \vec{r}_2) \vee .. \vee \vec{a} = R_k(\vec{\imath}) + \vec{r}_2)\}$$

où R_1, R_2, ..., R_k représentent des fonctions d'accès aux éléments de tableau dans le noyau original, les itérations $\vec{\imath}$ du nid de boucles de calculs sont définies par les bornes de boucles $B\vec{\imath} \leq \vec{b}$. Le calcul du meilleur Z-module R est discuté dans [4]. Les itérations $\vec{\imath}'$ correspondent aux itérations exprimées dans la nouvelle base de parcours et $C(\vec{\imath}')$ les contraintes après changement de base.
Cette nouvelle base de parcours dépend de l'émetteur ou du récepteur. Pour les processeurs émulant les bancs mémoire, les transferts s'effectuent selon l'ordre suivant : pour chaque banc mémoire, pour chaque processeur, pour chaque ligne du banc contenant un élément et enfin à partir de l'offset du premier élément de tableau de la ligne pour un nombre d'éléments donnés.
En réception, on parcourt les éléments en fonction du banc dont ils proviennent, dans l'ordre des dimensions du tableau en tenant compte de son *layout*. En C, les éléments de tableau sont stockés par ligne.

[4] Voir aussi l'architecture décrite dans [28].

Les parcours s'effectuent alors des dimensions les plus petites vers la dimension la plus grande pour laquelle les éléments sont contigus.
Il existe une relation de linéarisation entre le *layout* des éléments du tableau et leur placement sur les bancs mémoire que l'on peut exprimer par l'équation :

$$\sum_{i=1}^{n} a_i \cdot \prod_{k=i+1}^{n} \mathtt{dim}(a_k) = l \cdot NB \cdot LS + b \cdot LS + o$$

où $\mathtt{dim}(a_k)$ est le nombre d'éléments sur la dimension k, NB le nombre de bancs, LS la taille d'une ligne de banc, et o l'offset dans la ligne.
Opérateurs : projection, élimination de contraintes redondantes, génération de bornes de boucles

2.3.4. Compilation d'OpenMP vers MPI

L'outil STEP [39, 40] permet de transformer automatiquement un code écrit en Fortran OpenMP en un code équivalent écrit en Fortran MPI. Le code résultant est plus ou moins efficace suivant la précision avec laquelle les communications entre processeurs sont générées. Si les itérations parallèles sont réparties par blocs, les ensembles d'éléments de tableaux sont bien calculés statiquement. Si la répartition est cyclique, ils le sont beaucoup moins bien. L'outil STEP devrait être amélioré en utilisant des listes de polyèdres comme abstraction plutôt que de simples polyèdres.

2.3.5. Transferts entre hôte et GPU

L'optimisation et la génération des transferts de données entre la mémoire d'un processeur hôte et la mémoire d'une GPU via le bus PCIExpress nécessite le calcul et la prise en compte de plusieurs ensembles d'éléments de tableaux [3]. L'objectif est de ne transférer que les éléments utiles et de les transférer au meilleur moment pour ne ralentir ni la CPU ni la GPU.

2.3.6. Détermination des coefficients de tiling

Les coefficients de tiling doivent être choisis de manière à ce que l'ensemble des éléments de tableaux accédés lors du calcul d'une tuile tiennent soit dans la mémoire locale d'un accélérateur FPGA soit dans un cache. La connaissance de cet ensemble après un tiling symbolique permet d'en évaluer symboliquement le cardinal [13, 14] et d'en déduire numériquement la dimension qu'il faut utiliser pour les tuiles [29].
Opérateur : comptage symbolique des points entiers d'un polyèdre [13, 14]. En fait, il s'agit du calcul d'une fonction associant à un état mémoire σ le cardinal correspondant.

2.3.7. Vérification des accès aux tableaux

La vérification des accès peut se faire de deux manières différentes [44]. On peut soit protéger chaque accès par un test dont la condition permet de vérifier que les valeurs des expressions d'indices se trouvent bien dans les bornes des déclarations ou bien on peut s'appuyer sur les ensembles d'éléments tableaux accédés par une instruction complexe et vérifier a priori l'inclusion de ces ensembles dans les ensembles des éléments déclarés.
Avec la première stratégie, les tests redondants sont éliminés grâce à la technique décrite en section 2.1.3 ou à tout le moins simplifiés. Dans le deuxième cas, on peut montrer qu'aucun test n'est nécessaire ou bien générer un test minimal.
Opérateur : inclusion d'ensembles (inclusion de graphes de fonctions)

2.3.8. Conclusion sur les ensembles d'éléments de tableaux

Les ensembles d'éléments de tableaux interviennent dans de nombreuses situations. L'arrivée des accélérateurs de type GPU a déjà donné un nouvel élan à leur utilisation, tout en introduisant le besoin de nouvelles analyses. L'arrivée future de processeurs *manycore* devrait aussi stimuler la réutilisation et l'extension de ces techniques.

2.4. Ensemble de dépendances

Les ensembles de dépendance sont des contraintes entre itérations de boucles. Ces contraintes doivent être respectées par tous les ordonnancements des calculs élémentaires pour garantir que les résultats fonctionnels seront inchangés.

Dans le cadre du modèle polyédrique strict, les dépendances sont exprimées entre deux occurrences d'instructions, mais dans le cas des nids de boucle, il s'agit de deux itérations caractérisées par des vecteurs i et i' de même dimension, le nombre de boucles englobantes communes. On peut alors passer de l'ensemble des paires (i, i') à un ensemble plus simple de vecteurs de dépendance $d = i' - i$, tous lexicopositifs par définition de l'ordre d'exécution séquentiel.
De très nombreuses abstractions ont été proposées pour les ensembles de dépendances D. Leur précision est directement choisie en fonction de la transformation de programme à effectuer [55, 24].

2.4.1. Algorithme d'Allen et Kennedy

L'algorithme d'Allen et Kennedy utilise une abstraction très simple de l'ensemble des dépendances : les niveaux de dépendance [55, 24]. Un vecteur de dépendance d est dit de niveau k si toutes ses coordonnées d_i pour $i < k$ sont nulles et si $d_k > 0$. Chaque niveau définit un demi sous-espace et constitue un polyèdre convexe particulier. Et les niveaux de dépendance correspondent à une abstraction particulière d'un espace affine. Elle peut être calculée avec un test de dépendance présenté en section 3.3.

2.4.2. Méthode hyperplane

La méthode hyperplane, définie par L. Lamport [36], consiste à utiliser un ordonnancement affine monodimensionnel de direction h. Cet ordonnancement est valide si le sens des dépendances est respecté [30] :

$$\forall d \in D \; h.d \geq 1$$

La quantification sur D peut être simplifiée, quand D est approximé par un polyèdre convexe ne contenant pas de droites. On peut alors utiliser l'ensemble fini des rayons extrémaux R du cône de dépendance qui est formé par transitivité des dépendances contenues dans D et remplacer la condition précédente par :

$$hR \geq 1$$

L'ensemble des ordonnancements valides H est alors aussi un polyèdre convexe, dans lequel il faut choisir un ordonnancement particulier en fonction des valeurs des bornes de boucles, des expressions d'accès, des antémémoires et du parallélisme de la machine cible. Nous ne discutons pas ici du choix de la valeur optimale de h, bien qu'on puisse aussi faire intervenir les polyèdres dans les cas simples.
Une fois h déterminé, on peut construire une matrice unimodulaire pour effectuer toutes les itérations nécessaires dans une nouvelle base, ce qui nous ramène à la manipulation d'un ensemble d'itérations.
Opérateur : passage au dual

2.4.3. Légalité des transformations unimodulaires de boucles

La légalité d'une transformation unimodulaire de boucles définie par une matrice M consiste à vérifier que toutes les dépendances initiales restent des dépendances dans le code transformé. Par définition des dépendances, $d = i' - i$, les dépendances transformées sont donc de la forme Md. Il faut donc vérifier que

$$\forall d \in D \; Md \succ 0$$

où $\succ$ représente l'ordre lexicographique. Comme la combinaison linéaire positive de vecteurs lexicographique positifs est positive, il suffit de vérifier la condition précédente sur les rayons du cône de dépendance associé à D, $MR \succ 0$.
Opérateur : passage au dual, élimination de quantificateur

2.4.4. Légalité du tiling

Un tiling d'un nid de boucles est défini par un ensemble de partitionnements hyperplanaires [32, 8]. Chaque famille d'hyperplans est définie par une origine et par un vecteur normal h. L'ensemble de ces vecteurs peut être regroupé dans une matrice H. La légalité d'un tiling par rapport à un ensemble de dépendance D consiste à garantir qu'il existe un ordonnancement atomique des tuiles ne violant aucune dépendance. Une condition suffisante [31] est :

$$HR \geq 0$$

où R est à nouveau constitué des rayons du cône pointé défini par D. Comme le tiling est une généralisation des transformations unimodulaires, ce résultat n'est pas surprenant bien qu'il ne soit pas évident de voir l'échange de boucles comme un tiling...

2.4.5. Conclusion sur les dépendances

Les conditions de légalité des transformations de nids de boucles font intervenir l'ensemble des vecteurs de dépendance. La vérification de ces conditions nécessite une représentation finie de cet ensemble ou d'une sur-approximation. Le passage au dual est un moyen de reformuler ces conditions sans quantificateurs.

2.5. Conclusion sur les ensembles

Nous avons fait un tour rapide des ensembles utilisés dans un compilateur. Les ensembles de valeurs, d'éléments de tableaux, d'itérations et de dépendance peuvent tous être abstraits, de manière plus ou moins précise, par des polyèdres convexes qui permettent de les manipuler de manière finie même s'ils comportent un nombre non borné d'éléments. Rappelons que les communications peuvent nécessiter une information exacte.
Une fois étudiés les ensembles E et F, nous passons naturellement à leur produit cartésien, $E \times F$, et à l'ensemble de ses parties, les relations entre E et F.

3. Utilisations de relations

Les relations d'un ensemble vers un autre ont de nombreuses applications en parallélisation. Elles peuvent être représentées par leurs graphes, sous-ensembles des produits cartésiens des ensembles de départ et d'arrivée, ce qui permet d'utiliser les mêmes fonctions pour les traiter que pour les ensembles. Il faut néanmoins enrichir les fonctionnalités disponibles en ajoutant un opérateur pour combiner deux relations ou appliquer une relation à un ensemble.
Deux types de relations sont utilisées : les relations entre états mémoire, i.e. entre ensembles de valeurs, et les relations entre itérations.

3.1. Relations entre ensembles de valeurs : *Transformers*

L'exécution d'une instruction permet de passer d'un état mémoire à un autre. Pour un programme déterministe, l'état final est une fonction de l'état initial. Mais si l'instruction est compliquée parce qu'elle comporte un appel de procédure ou bien si elle dépend d'éléments inconnus comme c'est le cas lors de la lecture d'une valeur au clavier, le graphe de la fonction est sur-approximé et on obtient une relation entre les deux états mémoire. Cette relation est appelée *transformer*, voire transformeur de prédicats si les états mémoire sont connus par des prédicats.
Comme les transformers portent sur le produit cartésien de l'état mémoire avec lui-même, il est de dimension double par rapport aux états-mémoires. C'est incompatible avec les abstractions non relationnelles et les abstractions relationnelles limitées comme, par exemple, les octogones [41] ne sont pas toujours satisfaisantes. Par contre, les transformers permettent d'encapsuler tous les effets d'une fonction et de ne l'analyser qu'une seule fois, plutôt qu'une fois par site d'appel.
Opérateurs : combinaison, application, fermeture transitive, domaine, image.

3.2. Analyse interprocédurale : résumé des appels de procédures

L'utilisation première des *transformers* est de permettre le remplacement des appels de procédures par un changement d'état unique, un résumé des effets de la procédure. Ce résumé est construit de bas en haut sur le graphe des appels, en l'absence d'appels récursifs, et sur l'arbre syntaxique, quand le programme est structuré.
En première analyse, les transformers se propagent par composition de relations pour les séquences, $T_{S_1;S_2} = T_{S_1} \circ T_{S_2}$, par intersections et enveloppes convexes pour les tests et leurs conditions, et par fermeture transitive [6] pour les boucles. Les entrées et sorties de blocs qui modifient les *scopes* se traduisent par des expansions cylindriques et des projections. Les appels de procédures utilisent extensions

cylindriques, intersections et projections. Les graphes de flot de contrôle sont pour le moment traités en les restructurant avec l'algorithme de Bourdoncle [11], mais ce n'est pas la meilleure solution [6].
Les transformers ont été initialement introduits dans PIPS [34] pour réduire les temps d'analyse liés aux élargissements successifs nécessités par les boucles imbriquées, fréquentes en calcul scientifique. Leur calcul est présenté dans [33].
Depuis, ce calcul a été modifié pour améliorer leur précision en prenant en compte les préconditions disponibles au plus tôt, pour pouvoir les calculer itérativement et pour accepter des effets de bord dans les expressions. À la n-ième itération, le transformer de l'instruction S_i est calculé par la formule :

$$T^n_{S_i} = \mathcal{T}(S_i, P^{n-1}_{S_i}) \wedge P^{n-1}_{S_i}$$

en prenant $P^0_{S_i} = \mathrm{Id}$ à la première itération. La fonction $\mathcal{T}$ transforme une instruction en une relation polyédrique tout en prendant en compte la précondition courante ce qui permet de mieux linéariser certaines expressions comme, par exemple, les multiplications de deux variables. Une fois les transformers calculés, on propage les préconditions de la section 2.1.1 sur l'arbre de syntaxe abstrait et sur les graphes de flots de contrôle :

$$P^n_{S_i} = T^n_{S_{i-1}} \circ P^n_{S_{i-1}}$$

Néanmoins, les opérateurs polyédriques de base sont inchangés.
Opérateurs : projection, expansion cylindrique (ajout d'une nouvelle dimension), combinaison, intersection, image,... S'y ajoutent implicitement deux opérateurs très délicats et importants, l'élimination de redondance pour contrôler la complexité (simplexe, Fourier-Motzkin, Janus[49]), et la normalisation pour faciliter les tests de non-régression (formes de Smith, Hermite, tri lexicographique,...).

3.2.1. Récupération des boucles *for*

La restructuration de graphes de contrôle, à partir de code source ou bien de code assembleur ou binaire, consiste traditionnellement à faire apparaître des boucles `while`. Mais la parallélisation automatique se fonde sur les boucles `DO` qui n'en sont qu'un cas particulier.
L'algorithme de reconstruction des boucles `DO` est fondé sur les variables apparaissant dans la condition de la boucle `while`, sur le transformer du corps de boucle pour déterminer si et comment elles sont incrémentées et sur la précondition de boucle pour connaître la valeur initiale.

3.3. Relations entre itérations : test de dépendance

Les tests de dépendance consistent à déterminer si deux références r_1 et r_2 a un même tableau[5], placées dans deux instructions S_1 et S_2, calculées dans deux états mémoire distincts mais reliés entre eux, peuvent correspondre à une même case mémoire. En reprenant les notations des sections précédentes et en ajoutant i l'ensemble des indices de boucles communes aux deux instructions, on s'intéresse à l'ensemble :

$$\{\phi | \exists \sigma_1 \in P_{S_1}\ \exists \sigma_2 \in P_{S_2}\ \phi = \mathcal{E}(r_1, \sigma_1) \wedge \phi = \mathcal{E}(r_2, \sigma_2) \wedge T_{S_1,S_2}(\sigma_1, \sigma_2) \wedge \sigma_1(i) \prec \sigma_2(i)\}$$

où apparaît un transformer inhabituel, T_{S_1,S_2}, qui relie les états mémoire possibles avant S_1 à ceux de S_2. Ce transformer peut être calculé spécifiquement pour les tests de dépendances, mais son calcul n'est pas indispensable parce que l'ensemble des dépendances peut être sur-approximé pour tout calcul d'ordonnancement, dont la parallélisation.
Comme les préconditions sont approximatives, l'ensemble ci-dessus peut être remplacé par :

$$\{\phi | \exists \sigma_1 \in P_{S_1}\ \exists \sigma_2 \in P_{S_2}\ \phi \in \mathcal{E}(r_1, \sigma_1) \wedge \phi \in \mathcal{E}(r_2, \sigma_2) \wedge T_{S_1,S_2}(\sigma_1, \sigma_2) \wedge \sigma_1(i) \prec \sigma_2(i)\}$$

quand les expressions d'indices ne sont pas affines. En présence d'appels de procédures, les références peuvent aussi être remplacées par les régions convexes de tableaux :

$$\{\phi | \exists \sigma_1 \in P_{S_1}\ \exists \sigma_2 \in P_{S_2}\ \phi \in R_{S_1}(\sigma_1) \wedge \phi \in R_{S_2}(\sigma_2) \wedge T_{S_1,S_2}(\sigma_1, \sigma_2) \wedge \sigma_1(i) \prec \sigma_2(i)\}$$

[5] Pour être complet, il faudrait préciser si ces références correspondent à des lectures ou des écritures mémoire.

La projection de Fourier-Motzkin peut être utilisée pour décider si ces ensembles sont vides et sous quelle condition afin de calculer une abstraction des dépendances $d = \sigma_2(i) - \sigma_1(i)$ ou des relations entre itérations, $(\sigma_1(i), \sigma_2(i))$.

3.4. Utilisation des transformers pour le calcul des régions de tableaux

Les transformers sont largement utilisés dans les différents algorithmes de calcul de régions de tableaux. Ceci est présenté dans les sections 4.1.1, 4.1.2 et 4.1.5.

3.5. Conclusion sur les relations

Nous avons présenté seulement deux cas de relations, les relations entre états mémoire et les relations entre itérations, mais elles ont de multiples applications en optimisation et en parallélisation. Cependant il est parfois plus utile de manipuler des fonctions, comme nous le voyons dans la section suivante.

4. Utilisation de fonctions

Les fonctions sont des cas particuliers de relations, tout comme les relations sont des cas particuliers d'ensembles. Mais savoir qu'on dispose d'une relation de un vers un permet parfois de prendre des décisions clés. Les fonctions qui sont utilisées en parallélisation interprocédurale, en transformations dites *gros grain* et en réallocation mémoire sont des fonctions allant d'un état mémoire vers un ensemble d'éléments de tableaux[6] : il s'agit des divers types de régions convexes de tableaux. Le modèle polyédrique strict utilise des fonctions de dépendance exactes allant d'une itération vers une autre itération.

4.1. Fonctions vers des ensembles d'éléments de tableaux

Afin de paralléliser des programmes comportant des appels de fonctions tout en conservant une complexité raisonnable, une solution [50, 21] est d'abstraire le comportement de ces fonctions par des résumés des ensembles d'éléments de tableaux lus ou écrits (régions de tableaux READ et WRITE), qui sont ensuite traduits au niveau des sites d'appels en prenant en compte les relations entre les paramètres formels et réels. Ces ensembles doivent bien souvent être exprimés en tenant compte des valeurs des variables utilisées dans les expressions d'indice, et sont donc *fonction* de l'état-mémoire. On peut généraliser cette approche à toutes les structures de l'arbre syntaxique du langage (expression, affectation, séquence d'instructions, boucles, tests,...), ce qui permet aux analyses clientes d'effectuer un traitement uniforme indépendamment du type d'instruction considéré.
Ainsi, pour un élément l d'un langage $\tilde{\mathcal{L}}$, une région $\mathcal{R}$ d'un tableau T à n dimensions est une fonction de l'ensemble des état-mémoire Σ vers un ensemble de d'éléments de T qui peuvent être vus comme des coordonnées dans $\mathbb{Z}^n$. Chaque dimension du tableau est représentée par sa coordonnée Φ_i^T pour la i-ème dimension de T ; le vecteur des Φ_i^T est noté Φ^T :

$$\begin{aligned} \mathcal{R} : \quad & \tilde{\mathcal{L}} \longrightarrow (\Sigma \longrightarrow \wp(\mathbb{Z}^n)) \\ & l \longrightarrow \mathcal{R}[\![l]\!] = \lambda\sigma.\{\Phi^T : r(\Phi^T, \sigma)\} \end{aligned}$$

r exprimant la relation entre les coordonnées Φ^T des éléments de la région du tableau T et les valeurs des variables du programme dans l'état mémoire σ. Dans PIPS, r est représenté sous forme de polyèdre convexe.

> Par exemple, la région contenant l'élément `t[i]` peut être représentée par $\lambda\sigma.\{\Phi_1^t : \Phi_1^t = \sigma(i)\}, \sigma(i)$ étant la valeur de la variable `i` dans l'état mémoire σ.

Si les analyses de régions READ et WRITE sont principalement utilisées pour la parallélisation à gros grain des boucles (section 4.1.3), d'autres transformations de code comme la privatisation de tableaux (section 4.1.6) ou la génération des communications dans le cadre de machines à mémoire distribuée (section 2.3.1), nécessitent une connaissance des régions de tableaux importées et exportées par la portion de code considérée. Ce sont les régions IN et OUT qui sont présentées dans la section 4.1.5. Comme nous le verrons alors, les fonctions qui définissent ces analyses mettent en oeuvre un opérateur de différence, et reposent donc sur le calcul de sous-approximations qui posent des problèmes que nous abordons dans la section 4.1.1.

[6] Il est plus facile de les manipuler comme des fonctions vers des ensembles de $\mathcal{P}(\Phi)$ plutôt que des relations vers les éléments de Φ.

4.1.1. Sur- et sous-approximations

Que ce soit par manque d'informations disponibles lors de l'analyse ou parce que les solutions ne sont pas calculables en un temps fini, il est généralement impossible de représenter de manière exacte les résultats des analyses sémantiques, et il est nécessaire de recourir à des approximations.

Ainsi, dans PIPS, les sur-approximations des ensembles d'éléments de tableaux sont calculées sur le treillis des polyèdres convexes, l'opérateur d'union étant l'enveloppe convexe, qui définit une unique *meilleure* sur-approximation de l'union de deux polyèdres convexes.

Malheureusement, il n'existe pas d'opérateur définissant une unique *meilleure* sous-approximation de l'union de deux polyèdres convexes[7]. Ce problème est connu dans le domaine de l'interprétation abstraite [11, 18], et pas seulement pour la sous-approximation des ensembles à coordonnées dans $\mathbb{Z}$.

La solution que nous avons proposée [20, 22] repose sur le constat qu'il existe des critères pour déterminer l'exactitude de la sur-approximation. Ainsi, si l'on peut décider que la sur-approximation est en réalité une représentation exacte, alors elle peut aussi être utilisée comme sous-approximation, sinon on choisit l'ensemble vide comme sous-approximation. De ce fait, seule la sur-approximation est réellement calculée, mais un test d'exactitude doit être effectué à chaque étape du calcul. Lorsqu'il est vrai, on parle alors de région *exacte*.

Notre bibliothèque *C3 Linear* fournit les outils nécessaires pour tester l'exactitude de l'enveloppe convexe de deux polyèdres (c'est-à-dire lorsqu'elle est égale à l'union ensembliste), ainsi que l'exactitude de la projection le long d'une dimension (élimination d'une variable), qui sont, nous le verrons par la suite, les deux opérations principalement utilisées lors des analyses de régions exactes. Mais il est également nécessaire de pouvoir tester l'exactitude de l'abstraction sous forme de polyèdre convexe d'une expression, que ce soit un indice de tableau, une borne ou un incrément de boucle, ou encore la condition d'un test.

Opérateurs : enveloppe convexe exacte, projection exacte vis-à-vis des points entiers inclus dans les polyèdres rationnels

4.1.2. Régions convexes de tableaux lues et écrites

Les régions READ et WRITE représentent les effets du programme sur les variables [50] : elles contiennent les variables et éléments de tableaux lus ou écrits par les instructions (simples ou composées) lors de leur exécution. Ces deux types d'analyses étant très similaires, nous ne décrivons ici que l'analyse des régions READ.

$\mathcal{R}_r$, $\overline{\mathcal{R}}_r$ et $\underline{\mathcal{R}}_r$ définissent les sémantiques exactes et approchées des régions READ. Ces sémantiques sont définies en détail pour le langage Fortran dans [22], aussi bien pour les expressions que pour les instructions et sont également implémentées dans PIPS pour le langage C. Nous donnons ci-dessous le détail des plus importantes.

Expressions Nous ne considérons ici que les expressions sans effet de bord, leurs effets en lecture sont simplement l'union des effets en lecture de chacune des références mémoire. Pour des variables scalaires ou des tableaux, en utilisant la syntaxe du langage C, nous obtenons :

$$\begin{array}{lcl}
\mathcal{R}_r[\![const]\!] & = & \lambda\sigma.\varnothing \\
\mathcal{R}_r[\![var]\!] & = & \lambda\sigma.\{var\} \\
\mathcal{R}_r[\![var(exp_1,\ldots,exp_k)]\!] & = & \lambda\sigma.\{var[\phi_1]\ldots[\phi_k] : \phi_1 = \sigma(exp_1) \wedge \cdots \wedge \phi_k = \sigma(exp_k)\} \\
 & & \bigcup_{i=1..k} \mathcal{R}_r[\![exp_i]\!]
\end{array}$$

Pour les constantes et les variables scalaires, la représentation sous forme de région est toujours exacte. Pour les références à des éléments de tableaux, elle est exacte uniquement lorsque toutes les expressions d'indice peuvent être représentées par des expressions linéaires en fonction des valeur des variables du programme.

Affectation Nous considérons ici des affectations sans effets de bord dans la partie droite ou dans les expressions d'indice de la partie gauche. La partie gauche écrit la variable ou l'élément de tableau

[7] Pour s'en convaincre, il suffit de considérer les ensembles constitués respectivement des éléments de tableaux `t[1]` et `t[3]`. On peut définir plusieurs sous-approximation de leur union : `{t[1]}`, `{t[3]}` ou encore l'ensemble vide, sans que l'on puisse qualifier l'une de meilleure que l'autre.

référencé, mais a des effets en lecture sur les indices de tableau :

$$\begin{array}{lcl} \mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\ \mathcal{R}_r[\![var[exp_1]\dots[exp_k] = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \cup (\bigcup_{i=1..k} \mathcal{R}_r[\![exp_i]\!]) \end{array}$$

Les sur-approximations correspondantes sont obtenues en utilisant les opérateurs abstraits approchés à la place des opérateurs exacts :

$$\begin{array}{lcl} \overline{\mathcal{R}}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\ \overline{\mathcal{R}}_r[\![var[exp_1]\dots[exp_k] = exp]\!] & = & \overline{\mathcal{R}}_r[\![exp]\!] \,\overline{\cup}\, (\overline{\bigcup}_{i=1..k} \overline{\mathcal{R}}_r[\![exp_i]\!]) \end{array}$$

où $\overline{\cup}$ est l'opérateur d'enveloppe convexe. Pour la dernière expression, la sous-approximation est égale à la sur-approximation si $\overline{\mathcal{R}}_r[\![exp]\!]$ et toutes les $\overline{\mathcal{R}}_r[\![exp_i]\!]$ sont des régions *exactes* et si le test d'exactitude de leur enveloppe convexe est exacte.

Séquence Les effets en lecture de $S_1; S_2$ sont l'union des effets de S_1 et de ceux de S_2[8]. Toutefois, il faut tenir compte du fait que $\mathcal{R}_r[\![S_1]\!]$ (tout comme $\mathcal{R}_r[\![S_1; S_2]\!]$) est décrite en fonction des valeurs des variables dans l'état-mémoire précédant S_1, alors que $\mathcal{R}_r[\![S_2]\!]$ est décrite en fonction des valeurs des variables dans l'état-mémoire précédant S_2. Si $\mathcal{T}[\![S_1]\!]$ modélise la transformation de l'état-mémoire par l'exécution de S_1, alors les sémantiques exactes et surestimées des régions READ sont définies par[9] :

$$\begin{array}{lcl} \mathcal{R}_r[\![S_1; S_2]\!] & = & \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] \\ \overline{\mathcal{R}}_r[\![S_1; S_2]\!] & = & \overline{\mathcal{R}}_r[\![S_1]\!] \,\overline{\cup}\, \overline{\mathcal{R}}_r[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!] \end{array}$$

La définition de la sur-approximation nécessite quelques explications. En effet, l'approximation de la transformation de l'état-mémoire par l'exécution de S_1, présentée précédemment sous le nom de *transformer*, est définie comme une relation $\bar{t}$ entre les états-mémoire d'entrée et de sortie de l'instruction, et ne peut donc se composer directement avec une fonction. On définit donc par extension la fonction associée $\overline{\mathcal{T}}$, qui, pour une instruction donnée, fait correspondre à un état-mémoire un ensemble d'états-mémoire de sortie possibles :

$$\begin{array}{rl} \overline{\mathcal{T}} : & \mathbf{S} \longrightarrow (\mathbf{\Sigma} \longrightarrow \tilde{\wp}(\mathbf{\Sigma})) \\ & S \longrightarrow \lambda\sigma.\{\sigma' : \bar{t}(\sigma, \sigma')\} \end{array}$$

Toutefois, le domaine de sortie de cette fonction ($\tilde{\wp}(\mathbf{\Sigma})$) n'est pas le même que le domaine d'entrée des régions (Σ). Pour obtenir une sur-approximation, il suffit de composer la région considérée avec chacun des éléments de l'ensemble donné par $\overline{\mathcal{T}}$, puis d'en prendre l'union :

$$\overline{\mathcal{R}} \,\overline{\bullet}\, \overline{\mathcal{T}} = \lambda\sigma. \overline{\bigcup_{\sigma' \in \overline{\mathcal{T}}(\sigma)}} \overline{\mathcal{R}}(\sigma')$$

ce qui se ramène finalement à une projection :

$$\overline{\mathcal{R}} \,\overline{\bullet}\, \overline{\mathcal{T}} = \lambda\sigma. \operatorname{proj}_{\sigma'}\left(\{\Phi : \bar{t}(\sigma, \sigma') \wedge \bar{r}(\Phi, \sigma')\}\right)$$

Boucles `for` ou boucle `DO` Fortran Les boucles `for` du langage C semblables aux boucles `DO` du langage Fortran ont des propriétés de régularité qui permettent d'en étudier plus facilement le parallélisme. C'est traditionnellement sur l'analyse de ces boucles que les efforts d'analyse se sont naturellement focalisés dans PIPS.

Ainsi, pour une boucle de la forme `for(i=0; i<n;i++) S;` si $\overline{\mathcal{R}}_r[\![S]\!] = \lambda\sigma\{\phi : r_S(\phi, \sigma)\}$ est la région du corps de la boucle, et si t_{for} donne une sur-approximation de la relation entre l'état-mémoire avant exécution de la boucle et l'état-mémoire avant une quelconque exécution du corps

[8] Nous supposons ici que le programme ne s'arrête pas lors de l'exécution de S_1.

[9] La région lue $\mathcal{R}_r[\![S_1]\!]$ est dénoté par R_{S_1} et le transformer $\mathcal{T}[\![S_1]\!]$ par T_{S_1} quand ils sont utilisés et non calculés dans les autres sections.

de la boucle, alors une sur-approximation des régions lues par la boucle peut s'exprimer de la façon suivante :

$$\overline{\mathcal{R}}_r[\![\texttt{for}(i=0;i<n;i++)\quad S;]\!] = \overline{\mathcal{R}}_r[\![n]\!] \,\overline{\cup}\, \overline{\text{proj}}_k\left(\overline{\text{proj}}_{\sigma'}(\{\phi : r_S(\phi,\sigma') \wedge t_{for}(\sigma,\sigma') \wedge k = \sigma(i) \wedge 0 \leq k < \sigma(n)\})\right)$$

Ainsi, les équations définissant le calcul des approximations de régions de tableaux peuvent se réduire à des successions d'enveloppes convexes et de projection, opérations toutes implémentées dans la bibliothèque *C3 Linear* et pour lesquelles nous disposons de tests d'exactitude. Les résultats ainsi obtenus sont ensuite utilisés dans plusieurs autres phases d'analyses et de transformations. Nous détaillons ci-après les algorithmes de parallélisation et fusion à gros grain, le calcul des régions IN et OUT, ainsi que la privatisation et la scalarisation de tableaux.
Opérateurs : introduction des opérateurs sur-approximés

4.1.3. Algorithme de parallélisation à gros grain

Une fois qu'on dispose des régions de tableaux lues et écrites pour n'importe quelle instruction, on peut utiliser les conditions de Bernstein directement sur deux itérations du corps de boucle à condition de les étendre aux cas des régions de tableaux. La boucle est dite parallèle si aucune itération ne lit un élément de tableau écrit par une autre itération, et si aucune itération n'écrit un élément de tableau écrit par une autre itération.
La condition interdisant l'écriture d'une cellule mémoire par une itération postérieure à celle qui la lit s'écrit pour n'importe quelle variable v :

$$\{\phi |\ \exists \sigma, \sigma' \in P_B\ \ \phi \in R_{B,v}(\sigma) \wedge \phi \in W_{B,v}(\sigma') \wedge \sigma(i) < \sigma'(i)\} = \emptyset$$

ou plus précisément :

$$\{\phi |\ \exists \sigma, \sigma' \in P_B\ \ \phi \in R_{B,v}(\sigma) \wedge \phi \in W_{B,v}(\sigma') \wedge T_{B,B}(\sigma,\sigma')\} = \emptyset$$

où $T_{B,B}$ représente le déroulement d'au moins une itération, soit la fermeture transitive d'une itération $T_{B,B} = T_B^+$, en admettant que B incorpore le passage à l'itération suivante. Les deux autres conditions de Bernstein s'écrivent de manière similaire.
Cet algorithme de parallélisation est très simple, voire simpliste, par rapport à celui d'Allen et Kennedy, mais il permet de traiter n'importe quel corps de boucle, sans restriction sur le contrôle ou les appels de procédures, il évite des distributions de boucles inutiles et améliore donc la localité et la taille des boucles. Pour qu'il soit vraiment efficace, il faut néanmoins le compléter pour privatiser les tableaux intermédiaires et pour détecter les réductions.
De plus, cet algorithme pourrait utiliser les régions importées et exportées introduites dans la section suivante afin d'incorporer directement la privatisation de sections de tableaux dans le processus de parallélisation.

4.1.4. Fusion de boucles à gros grain

Lors d'un exposé présenté à l'un des workshops de CGO en 2011, David Monniaux nous a demandé s'il serait possible d'utiliser la même technique pour effectuer une fusion de boucles, l'algorithme standard étant fondé sur des propriétés des graphes de dépendances avant et après fusion [23]. La réponse nous semble devoir être positive.
La fusion de deux boucles adjacentes[10] est légale si les dernières itérations de la première boucle n'écrivent pas des données lues ou écrites par les premières itérations de la deuxième boucle et si elles ne lisent pas des données écrites par ces mêmes premières itérations.
En supposant que les indices des deux boucles soit i_1 et i_2 et leurs corps B_1 et B_2 et en supposant qu'elles puissent être fusionnées avec un délai d, c'est-à-dire que l'itération $\sigma_1(i_1)$ soit exécutées en même temps

[10] Les régions de tableaux peuvent aussi être utilisées pour décider si des instructions situées entre les deux boucles peuvent être déplacées pour les rendre adjacentes. Dans tous les cas, il s'agit de déterminer si deux instructions, aussi complexes soient-elles, peuvent être permutées sans modifier les résultats du programme.

que l'itération $\sigma_2(i_2)$ vérifiant $\sigma_1(i_1) = \sigma_2(i_2) + d$, la première condition peut s'écrire pour n'importe quelle variable v modifiées dans la première boucle :

$$\{\phi | \exists \sigma_1 \in P_{B_1} \exists \sigma_2 \in P_{B_2} \phi \in W_{B_1,v}(\sigma_1) \cap R_{B_2,v}(\sigma_2) \wedge \sigma_1(i_1) > \sigma_2(i_2) + d \wedge T_{B_1,B_2}(\sigma_1, \sigma_2)\} = \emptyset$$

de manière très similaire à ce qui est fait pour la parallélisation.
On remarque d'abord que le transformer $T_{B_1,B_2}(\sigma_1, \sigma_2)$ n'est pas disponible de manière standard, mais que son omission ou son imprécision ne perturbe pas la validité du test. On remarque ensuite que les domaines d'itérations ne sont pas pris en compte ici et que le problème est déplacé sur la phase de génération de code. On remarque enfin que les régions R et W peuvent être remplacées par des régions IN et OUT si les problèmes de privatisation ou de renommage sont pris en compte par la phase de génération de code. Il reste néanmoins à minimiser d.

4.1.5. Régions convexes de tableaux importées et exportées

Les régions IN et OUT contiennent les variables scalaires et les ensembles d'éléments de tableaux dont la valeur est respectivement *importée* ou *exportée* par le fragment de code considéré. Ces définitions ne sont toutefois pas symétriques : les régions IN ne dépendent pas des régions qui ont été initialisées[11] avant l'exécution des instructions précédant le fragment de code considéré, et sont calculées depuis les feuilles de l'arbre de syntaxe abstrait (AST) vers sa racine. Au contraire, les régions OUT dépendent des régions importées par la suite du programme et sont calculées depuis la racine de l'AST vers les feuilles. Nous présentons tout d'abord le calcul des régions IN puis celui des régions OUT[12].
Pour un langage dont les expressions n'ont pas d'effet de bord, les régions IN sont initialisées au niveau des affectations :

Affectation En C comme en Fortran, la partie droite d'une affectation est évaluée avant la partie gauche. La valeur des éléments référencés en lecture par la partie droite ne peut donc venir de la partie gauche, et ces valeur sont ainsi nécessairement importées. Comme nos expressions n'ont pas d'effet de bord en Fortran[13], les régions IN d'une affectation sont donc ses régions READ :

$$\mathcal{R}_i[\![ref = exp]\!] = \mathcal{R}_r[\![ref = exp]\!]$$

Les approximations sont directement dérivées de cette définition.

Séquence Les régions IN de la séquence $S_1; S_2$ contiennent les éléments de tableaux importés par S_1, $\mathcal{R}_i[\![S_1]\!]$[14], plus ceux importés par S_2 après l'exécution de S_1 ($\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$) mais qui ne sont pas définis par S_1 ($\mathcal{R}_w[\![S_1]\!]$) :

$$\mathcal{R}_i[\![S_1; S_2]\!] = \mathcal{R}_i[\![S_1]\!] \cup \big((\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \boxminus \mathcal{R}_w[\![S_1]\!]\big)$$

L'opérateur $\boxminus$ désigne ici la différence ensembliste, qui nécessite quelques précautions pour la définition de la sur-approximation. En effet, une sur-approximation d'une différence de deux ensembles ne peut pas être égale à la différence de leurs sur-approximations, mais doit être définie comme la différence d'une sur-approximation du premier ensemble et d'une sous-approximation du deuxième ensemble. Une difficulté supplémentaire provient du fait que la différence de deux polyèdres convexes n'est pas nécessairement un polyèdre convexe, et donc que l'opérateur de différence doit lui aussi être surestimé ($\overline{\boxminus}$) :

$$\overline{\mathcal{R}}_i[\![S_1; S_2]\!] = \overline{\mathcal{R}}_i[\![S_1]\!] \,\overline{\cup}\big((\overline{\mathcal{R}}_i[\![S_2]\!] \,\overline{\bullet}\, \overline{\mathcal{T}}[\![S_1]\!]) \,\overline{\boxminus}\, \underline{\mathcal{R}}_w[\![S_1]\!]\big)$$

Dans notre implémentation, nous calculons tout d'abord une représentation exacte de la différence sous forme d'une disjonction de polyèdres convexes [37, 38], dont nous calculons ensuite l'enveloppe convexe afin de conserver une représentation compacte sous la forme d'un unique polyèdre. Si la disjonction ne contient qu'un seul élément, alors la différence est considérée comme exacte.

[11] Ou non, si on veut détecter les variables non initializées [45].
[12] Les régions OUT sont calculées de manière interprocédurale, mais on peut aussi les calculer intraprocéduralement.
[13] Les notations sont plus complexes pour C, à moins que les effets de bords soient déplacés par une autre passe.
[14] La région $\mathcal{R}_i[\![S_1]\!]$ est notée IN_{S_1} quand elle est utilisée et non calculée dans les autres sections.

Boucle `for` ou boucle `do` fortran La contribution d'une itération k aux régions IN d'une boucle de ce type est égale aux régions IN de cette itération auxquelles il faut enlever les régions écrites dans les itérations k' précédentes ($1 \leq k' < k$). Comme nous l'avons vu précédemment pour les régions READ, il faut en outre tenir compte de la modification de l'état-mémoire par chacune des itérations. Nous désignons à nouveau par t_{for} le polyèdre convexe décrivant la relation entre l'état-mémoire avant l'exécution de la boucle, et l'état-mémoire précédant n'importe quelle exécution du corps de la boucle. L'équation suivante donne alors la sur-approximation des régions IN pour l'ensemble de la boucle :

$$\begin{aligned}&\overline{\mathcal{R}}_i[\![\texttt{for}(i=0;i<n;i++)\quad S;]\!] = \\ &\quad \overline{\mathcal{R}}_r[\![n]\!] \quad \overline{\cup} \quad \overline{\text{proj}}_k\Big(\overline{\text{proj}}_{\sigma'}\big(\{\phi : \bar{r}_S(\phi,\sigma') \wedge t_{for}(\sigma,\sigma') \wedge k = \sigma(i) \wedge 1 \leq k \leq \sigma(n)\}\big) \\ &\qquad\qquad \overline{\boxminus} \quad \underline{\text{proj}}_{k'}\big(\underline{\text{proj}}_{\sigma'}(\{\phi : \underline{w}_S(\phi,\sigma') \wedge t_{for}(\sigma,\sigma') \wedge k' = \sigma(i) \wedge 1 \leq k' < k\})\big)\Big)\end{aligned}$$

où $\bar{r}_S(\phi,\sigma')$ est un polyèdre dérivant la sur-approximation de la région READ du corps de la boucle[15], et $\underline{w}_S(\phi,\sigma')$ le polyèdre décrivant la sous-approximation de la région WRITE du corps de la boucle.

Les régions OUT peuvent alors être définies en fonction des régions READ, WRITE et IN. Elles sont initialisées à l'ensemble vide pour l'ensemble du programme et sont propagées vers les instructions élémentaires, affectations ou appels de fonctions.

Instructions d'une séquence Nous supposons que les régions OUT de la séquence $S_1;S_2$ sont connues, et notre but est de calculer les régions OUT de S_1 et de S_2, respectivement $\mathcal{R}_o[\![S_1]\!]$ et $\mathcal{R}_o[\![S_2]\!]$.

Intuitivement, les régions exportées par S_2 sont les régions exportées par la séquence, et écrite par S_2. Il faut cependant faire attention à l'effet de S_1 sur l'état mémoire : pour $\mathcal{R}_o[\![S_1;S_2]\!]$, l'état de référence est celui précédant S_1 ; alors que pour $\mathcal{R}_o[\![S_2]\!]$, c'est celui précédant S_2, obtenu en appliquant $\mathcal{T}[\![S_1]\!]$. $\mathcal{R}_o[\![S_2]\!]$ doit donc vérifier :

$$\mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!] = \mathcal{R}_o[\![S_1;S_2]\!] \cap \mathcal{R}_w[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

Nous ne pouvons pas donner une définition constructive de $\mathcal{R}_o[\![S_2]\!]$ car $\mathcal{T}[\![S_1]\!]$ n'est pas inversible[16]. Mais à partir de la définition du *transformeur* sous forme de relation, on peut en définir une sur-approximation, que nous appelons abusivement *transformeur inverse* :

$$\overline{\mathcal{T}}^{-1}[\![S_1]\!] = \lambda\sigma.\{\sigma : t_{S_1}(\sigma',\sigma)\}$$

(noter les places relatives de σ et σ' qui sont inversées par rapport à la définition de $\overline{\mathcal{T}}[\![S_1]\!]$). Ceci nous permet de définir une sur-approximation de $\mathcal{R}_o[\![S_2]\!]$:

$$\overline{\mathcal{R}}_o[\![S_2]\!] = \overline{\mathcal{R}}_o[\![S_1;S_2]\!] \,\bar{\bullet}\, \overline{\mathcal{T}}^{-1}[\![S_1]\!] \cap \overline{\mathcal{R}}_w[\![S_2]\!]$$

La composition d'une région avec le transformeur inverse est définie similairement à la composition avec le transformeur. L'intersection ne nécessite pas de sur-approximation, car c'est une opération qui est toujours exacte dans l'ensemble des polyèdres convexes.

Les régions exportées par S_1 sont les régions exportées par la séquence, mais pas par S_2, et qui sont écrites par S_1. A ceci, il faut ajouter les éléments écrits par S_1 et exportés vers S_2, c'est-à-dire importés par S_2 :

$$\mathcal{R}_o[\![S_1]\!] = \mathcal{R}_w[\![S_1]\!] \cap \big((\mathcal{R}_o[\![S_1;S_2]\!] \boxminus \mathcal{R}_o[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \cup \mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]\big)$$

Ici, les approximations sont directement dérivées de la sémantique exacte.

[15] Ceci est équivalent à la notation fonctionnelle $\phi \in R_S(\sigma)$ utilisée par ailleurs.

[16] On ne peut pas retrouver l'état-mémoire d'entrée à partir de l'état-mémoire de sortie.

Corps d'une boucle `for` ou boucle `do` Fortran On suppose connues les régions OUT pour l'ensemble de la boucle ($\overline{\mathcal{R}}_o[\![\texttt{for}(i = 0; i < n; i++)S;]\!] = \lambda\sigma.\{\phi : \overline{r}_o(\phi, \sigma)\}$), et l'on cherche les régions OUT d'une itération k quelconque ($\overline{\mathcal{R}}_o[\![S]\!]_k$). Ce sont les régions écrites par cette itération, et qui sont exportées par la boucle, sans être écrites lors d'une itération k' suivante ; ou alors qui sont écrites par cette itération puis importées par une itération k' suivante, mais sans être écrite par une itération k'' intermédiaire ($k < k'' < k'$). Nous donnons ici directement l'équation permettant de calculer la sur-approximation à partir de la sur-approximation des régions IN du corps de la boucle ($\overline{\mathcal{R}}_i[\![S]\!] = \lambda\sigma.\{\phi : \overline{r}_i(\phi, \sigma)\}$), de la sous-approximation des régions WRITE du corps de la boucle ($\underline{\mathcal{R}}_w[\![S]\!] = \lambda\sigma.\{\phi : \underline{r}_w(\phi, \sigma)\}$) et ainsi que de la sur-approximation t du transformeur de la boucle (pris ici sous sa forme de relation) qui relie les état-mémoire de l'entrée de la boucle et de l'entrée d'une itération quelconque :

$$\begin{aligned}
\overline{\mathcal{R}}_o[\![S]\!]_k = \ & \overline{\mathcal{R}}_w[\![S]\!] \cap \Big(\Big(\overline{\text{proj}}_{\sigma'}\big(\lambda\sigma.\{\phi : \overline{r}_{in}(\phi,\sigma) \wedge t(\sigma',\sigma) \wedge k = \sigma(i)\} \big) \\
& \qquad \overline{\boxminus}\ \underline{\text{proj}}_{k'}\big(\underline{\text{proj}}_{\sigma'}\big(\lambda\sigma.\{\phi : \underline{r}_w(\phi,\sigma) \wedge t(\sigma,\sigma') \wedge k' = \sigma'(i)\} \big)\big) \Big) \\
\overline{\cup}\ \overline{\text{proj}}_{k'}\Big(& \overline{\text{proj}}_{\sigma'}\big(\lambda\sigma.\{\phi : \overline{r}_{in}(\phi,\sigma) \wedge t(\sigma,\sigma') \wedge k < k' \leq \sigma(n) \wedge k' = \sigma'(i)\} \big) \\
& \qquad \overline{\boxminus}\ \underline{\text{proj}}_{k''}\big(\underline{\text{proj}}_{\sigma'}\big(\lambda\sigma.\{\phi : \underline{r}_w(\phi,\sigma) \wedge t(\sigma,\sigma') \\
& \qquad\qquad \wedge k < k' \leq \sigma(n) \wedge k < k'' \leq k' - 1 \wedge k'' = \sigma'(i)\} \big)\big) \Big) \Big)
\end{aligned}$$

Tout comme pour les régions READ et WRITE les équations permettant le calcul des approximations de régions IN et OUT peuvent s'exprimer en fonction d'opérations simples sur les polyèdres convexes : enveloppe convexe, intersection et projection, et d'une opération de différence passant par l'intermédiaire des disjonctions de polyèdres. On peut aussi retenir que ces analyses réutilisent les résultats des analyses de *transformeurs*, et que nous avons dû définir les opérations permettant de les composer avec les régions.

4.1.6. Privatisation

Une variable v peut être privatisée dans une instruction si elle y est utile, si la ou les valeurs qu'elle possède en mémoire en entrée ne sont pas utilisées par l'instruction elle-même et si sa ou ses valeurs en sortie ne sont pas utilisées par la continuation. Le cas le plus commun de privatisation a lieu dans les corps de boucle. Si on dispose des régions de tableaux convexes, les fonctions $\text{IN}_{B,v}$ et $\text{OUT}_{B,v}$, v est donc utilement privatisable dans le corps de boucle B exécuté sous la précondition P_B si

$$\forall\sigma \in P_B\ W_{B,v}(\sigma) \neq \emptyset \wedge \text{IN}_{B,v}(\sigma) = \text{OUT}_{B,v}(\sigma) = \emptyset$$

4.1.7. Scalarisation

Soit B un corps de boucle et i son indice. Soit $W_{B,v}$ et $R_{B,v}$ les fonctions qui associent à l'état courant σ l'ensemble des éléments de v qui sont lus ou écrits dans B. On définit une relation R de l'ensemble des valeurs possibles de i vers les parties de l'ensemble des parties des éléments de v, $\mathcal{P}(\Phi)$

$$R : Val \to \mathcal{P}(\Phi) \text{ s.t. } R(v) = \{\phi | \exists\sigma\ \sigma(i) = v \wedge \phi \in W_{B,v}(\sigma) \cup R_{B,v}(\sigma)\}$$

Si R est une application, toutes les références à v se trouvant dans le corps de boucle peuvent être remplacées par des références à un scalaire temporaire puisque, par définition d'une fonction, chaque itération σ_i utilise un unique élément de v.

L'insertion d'une initialisation du temporaire ou bien d'une exportation de sa valeur en sortie de boucle est décidée en fonction des régions $\text{IN}_{B,k}$ et $\text{OUT}_{B,k}$. Le contenu du corps de boucle est totalement ignoré contrairement à ce qui est fait traditionnellement [12]. Mais on ne peut l'ignorer pour décider de la profitabilité de la transformation.

Opérateurs : déterminer si un graphe de relation est un graphe de fonction ou un graphe d'application.

4.2. Fonctions vers des ensembles d'itérations

L'information de dépendance utilisée pour calculer un ordonnancement affine multidimensionnel est une fonction affine par morceaux qui associe à une occurrence d'instruction et à une case mémoire lue par cette occurrence, une autre occurrence d'instruction qui a calculée la valeur contenue dans cette case mémoire [27].
Cette fonction n'est pas calculable en général, mais elle l'est pour un sous-ensemble de programmes appelés programmes à contrôle statique [37].

4.3. Conclusion sur les fonctions

L'importance des fonctions dans le compilateur PIPS a été réalisée progressivement au cours des vingt dernières années. Les API définies au niveau des polyèdres convexes n'ont malheureusement pas été complétées par des API au niveau des relations et des fonctions. Les structures de données de la bibliothèque *C3 Linear* sont donc manipulées directement dans le code de PIPS.

5. Conclusion

Nous avons montré que les abstractions issues de l'interprétation abstraite pouvaient être utilisées et réutilisées au sein d'un compilateur. L'abstraction polyédrique constitue un bon compromis entre précision et vitesse d'analyse. Elle bénéficie de fondements mathématiques solides. Elle permet d'encoder par des sur-approximations ensembles, fonctions et relations d'une manière uniforme.
Ceci suppose que nous disposions de librairies polyédriques complètes et robustes par rapport au temps d'exécution, à l'espace mémoire et à la magnitude des coefficients. Tous doivent être maîtrisés pour utiliser une telle bibliothèque au sein d'un compilateur. De nombreuses librairies polyédriques ont été écrites et ré-écrites [35, 42, 47, 51, 52, 53], avec des objectifs et des technologies différents, de manière collaborative ou non. Les algorithmes de base ont été progressivement améliorés, en particulier pour prendre en compte précisemment les ensembles de points entiers se trouvant dans les polyèdres convexes, et des benchmarks sont disponibles sur Internet[43]. Il reste encore à garantir la monotonie des résultats en cas d'exception : il ne faut pas qu'un ajout d'information en entrée conduise à une perte en sortie. Malgré les travaux de Duong Nguyen[43], cet objectif n'est pas encore atteint avec notre bibliothèque *C3 Linear*.
Une bibliothèque polyédrique ne suffit naturellement pas à écrire simplement un compilateur optimiseur. La profitabilité des transformations conduit à y adjoindre une bibliothèque pour manipuler des polynômes[56]. L'application des transformations peut être simplifiée par l'utilisation d'un outil de gestion de structures de données comme Newgen[16]. Enfin, la légalité de tous les algorithmes de transformations du middle-end n'a pas encore été réduite à des conditions polyédriques simples. Les graphes ont toujours un rôle important à jouer et les optimisations de l'évaluation des expressions s'appuient sur d'autres propriétés et transformations [57]. Le choix de la représentation intermédiaire est aussi critique.
La parallélisation automatique est-elle un échec comme le clame Keshav Pingali dans son exposé à LCPC 2010 parce qu'elle ne s'applique vraiment efficacement qu'au domaine du calcul scientifique dense et du traitement du signal ? Au pire, ce ne serait qu'un échec relatif puisque les objectifs visés initialement ont été atteints et que les techniques qui en ont découlé s'appliquent de plus en plus largement dans les compilateurs et les analyseurs statiques et qu'elles peuvent encore être étendues pour combiner analyses statiques et dynamiques, pour étendre les objectifs de la vitesse de calcul à l'énergie consommée et à la sûreté comme cela a été fait, par exemple, pour la vérification des accès aux tableaux, et pour profiter de nouvelles sources d'information comme les compteurs matériels.

Bibliographie

1. Alfred V. AHO, Monica LAM, Ravi SETHI et Jeffrey D. ULLMAN : *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
2. Alfred V. AHO, Ravi SETHI et Jeffrey D. ULLMAN : *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. Mehdi AMINI, François IRIGOIN et Ronan KERYELL : Compilation et optimisation statique des communications hôte-accélérateur. *In Rencontres Francophones du Parallélisme (RenPar)*, Saint-Malo, France, mai 2011.

4. Corinne ANCOURT : *Génération automatique de code de transfert pour multiprocesseurs à mémoires locales*. Thèse de doctorat, Université Pierre et Marie Curie (Paris 6), 1991.
5. Corinne ANCOURT, Fabien COELHO, Béatrice CREUSILLET et Ronan KERYELL : How to Add a New Phase in PIPS : the Case of Dead Code Elimination. *In Sixth Workshop on Compilers for Parallel Computers (CPC)*, pages 19–30, Aachen, Germany, décembre 1996.
6. Corinne ANCOURT, Fabien COELHO et François IRIGOIN : A Modular Static Analysis Approach to Affine Loop Invariants Detection. *In NSAD : 2nd International Workshop on Numerical and Symbolic Abstract Domains*, ENCTS, Perpignan, France, septembre 2010. Elsevier.
7. Corinne ANCOURT, Fabien COELHO, François IRIGOIN et Ronan KERYELL : A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, 6:3–27, 1997.
8. Corinne ANCOURT et François IRIGOIN : Scanning polyhedra with do loops. *In PPOPP*, pages 39–50, 1991.
9. Corinne ANCOURT et François IRIGOIN : Automatic code distribution. *In CPC*, 1992.
10. Cédric BASTOUL : Code generation in the polyhedral model is easier than you think. *In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, septembre 2004.
11. François BOURDONCLE : *Sémantiques des langages d'ordre supérieur et interprétation abstraite*. Thèse de doctorat, École polytechnique, novembre 1992.
12. Steve CARR et Ken KENNEDY : Scalar replacement in the presence of conditional control flow. *Softw. Pract. Exper.*, 24:51–77, January 1994.
13. Philippe CLAUSS : Counting solutions to linear and nonlinear constraints through Ehrhart polynomials : applications to analyze and transform scientific programs. *In Proceedings of the 10th international conference on Supercomputing*, ICS '96, pages 278–285, New York, NY, USA, 1996. ACM.
14. Philippe CLAUSS et Vincent LOECHNER : Parametric Analysis of Polyhedral Iteration Spaces. *J. VLSI Signal Process. Syst.*, 19:179–194, July 1998.
15. Fabien COELHO : *Contributions à la compilation du High Performance Fortran*. Thèse de doctorat, École des mines de Paris, October 1996.
16. Fabien COELHO, Pierre JOUVELOT, François IRIGOIN et Corinne ANCOURT : Data and Process Abstraction in PIPS Internal Representation. *In Workshop on Internal Representations (WIR)*, Chamonix, France, avril 2011.
17. Patrick COUSOT : Abstract interpretation. *ACM Comput. Surv.*, 28:324–328, June 1996.
18. Patrick COUSOT et Radhia COUSOT : Abstract interpretation framework. *Journal of Logic and Computation*, 2(5):511–547, août 1992.
19. Patrick COUSOT et Nicolas HALBWACHS : Automatic discovery of linear restraints among variables of a program. *In Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.
20. Béatrice CREUSILLET et François IRIGOIN : Exact vs. approximate array region analyses. *In Languages and Compilers for Parallel Computing*, numéro 1239 de Lecture Notes in Computer Science, pages 86–100. Springer-Verlag, août 1996.
21. Béatrice CREUSILLET et François IRIGOIN : Interprocedural array region analyses. *International Journal of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996.
22. Béatrice CREUSILLET-APVRILLE : *Analyses de régions de tableaux et applications*. Thèse de doctorat, École des mines de Paris, décembre 1996.
23. Alain DARTE : On the complexity of loop fusion. *In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 149–, Washington, DC, USA, 1999. IEEE Computer Society.
24. Alain DARTE et Frédéric VIVIEN : On the Optimality of Allen and Kennedy's Algorithm for Parallelism Detection in Nested Loops. *In Europar'96*, pages 379–388. Springer Verlag, 1993.
25. Paul FEAUTRIER : Some efficient solutions to the affine scheduling problem : I. One-dimensional time. *International Journal of Parallel Programming*, 21:313–348, octobre 1992.
26. Paul FEAUTRIER : Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992. 10.1007/BF01379404.
27. Paul FEAUTRIER : Parallélisation automatique, histoire et perspectives. *TSI*, pages 161–182, 2005.
28. Pierre FIORINI, François IRIGOIN et Ronan KERYELL : ModÃ¨le de compilation d'hpf pour la ma-

chine mimd Ã bancs mÃ©moire et rÃ©seau distribuÃ© programmable phÃ©nix. *In 8ème Rencontres Francophones du Parallélisme (RenPar'8)*, Bordeaux, France, mai 1996.
29. Serge GUELTON, Ronan KERYELL et François IRIGOIN : Compilation pour cibles hétérogènes : automatisation des analyses, transformations, et décisions nécessaires. *In Rencontres Francophones du Parallélisme (RenPar)*, Saint-Malo, France, mai 2011.
30. F. IRIGOIN : Loop reordering with dependence direction vectors. *In Journées Firtech systèmes et télématique, Architectures futures : programmation parallèle et intégration VLSI*, 1988.
31. F. IRIGOIN et R. TRIOLET : Supernode partitioning. *In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.
32. François IRIGOIN : *Partitionnement des boucles imbriquées. Une technique d'optimisation pour les programmes scientifiques*. Thèse de doctorat, Université Pierre et Marie Curie (Paris 6), June 1987.
33. François IRIGOIN : Interprocedural analyses for programming environments. *In Workshop on Environments and Tools For Parallel Scientific Computing, CNRS-NSF*, septembre 1992.
34. François IRIGOIN, Pierre JOUVELOT et Rémi TRIOLET : Semantical interprocedural parallelization : an overview of the PIPS project. *In Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM.
35. Bertrand JEANNET et Antoine MINÉ : Apron : A library of numerical abstract domains for static analysis. *In Computer Aided Verification*, volume 5643 de *Lecture Notes in Computer Science*, pages 661–667. Springer-Verlag, 2009.
36. Leslie LAMPORT : The hyperplane method for an array computer. *In Proceedings of the Sagamore Computer Conference on Parallel Processing*, pages 113–131, London, UK, 1975. Springer-Verlag.
37. Arnauld LESERVOT : Extension de C3 aux unions de polyèdres. Rapport technique, Commissariat à l'Énergie Atomique, janvier 1994.
38. Arnauld LESERVOT : *Analyses interprocédurales du flot des données*. Thèse de doctorat, Université Paris VI, mars 1996.
39. Daniel MILLOT, Alain MULLER, Christian PARROT et Frédérique SILBER-CHAUSSUMIER : STEP : a distributed OpenMP for coarse-grain parallelism tool. *In Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 83–99, 2008.
40. Daniel MILLOT, Alain MULLER, Christian PARROT et Frédérique SILBER-CHAUSSUMIER : From OpenMP to MPI : first experiments of the STEP source-to-source transformation. *In Mini-symposium in International Conference on Parallel Computing (ParCO) - Parallel Programming Tools for Multi-core Architectures*, 2009.
41. Antoine MINÉ : The octagon abstract domain. *In AST 2001 in WCRE 2001, IEEE*, pages 310–319. IEEE CS Press, 2001.
42. MINES-PARISTECH : Linear C3 Library of PIPS. http ://pips4u.org. Open source, under GPLv3.
43. Duong NGUYEN : *Robust and Generic Abstract Domain for Static Program Analyses : The Polyhedral Case*. Thèse de doctorat, École des mines de Paris, November 2010.
44. Thi Viet Nga NGUYEN et François IRIGOIN : Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27:527–570, May 2005.
45. Thi Viet Nga NGUYEN, François IRIGOIN, Corinne ANCOURT et Fabien COELHO : Automatic detection of uninitialized variables. *In Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 217–231, Berlin, Heidelberg, 2003. Springer-Verlag.
46. William PUGH : The Omega Test : a fast and practical integer programming algorithm for dependence analysis. *In Supercomputing '91 : Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.
47. William PUGH et the entire OMEGA PROJECT TEAM : The Omega library. http ://www.cs.umd.edu/projects/omega.
48. Alexander SCHRIJVER : *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.
49. Jean-Claude SOGNO : The janus test : A hierarchical algorithm for computing direction and distance vectors. *In Proceedings of the 29th Hawaii International Conference on System Sciences Volume 1 : Software Technology and Architecture*, HICSS '96, pages 203–, Washington, DC, USA, 1996. IEEE Computer Society.

50. Rémi TRIOLET : *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure*. Thèse de doctorat, Université Pierre et Marie Curie (Paris 6), June 1984.
51. UNIVERSITY OF PARMA, ITALY : The PPL : A Library for Representing Numerical Abstractions. http ://www.cs.unipr.it/ppl, 2004.
52. Sven VERDOOLAEGE : ISL : An Integer Set Library for the Polyhedral Model. http ://freshmeat.net/projects/isl, 2010.
53. Doran WILDE et ICPS : The Polylib library. http ://icps.u-strasbg.fr/polylib.
54. Yi-Qing YANG : *Tests des dépendances et transformations de programme*. Thèse de doctorat, Université Pierre et Marie Curie (Paris 6), novembre 1993.
55. Yi-Qing YANG, Corinne ANCOURT et François IRIGOIN : Minimal data dependence abstractions for loop transformations : extended version. *Int. J. Parallel Program.*, 23:359–388, August 1995.
56. Lei ZHOU : *Analyse statique et dynamique de la complexité des programmes scientifiques*. Thèse de doctorat, Université Pierre et Marie Curie (Paris 6), septembre 1994.
57. Julien ZORY : *Contributions à l'optimisation de programmes scientifiques*. Thèse de doctorat, École des mines de Paris, décembre 1999.