UNIVERSITÉ LOUIS PASTEUR
STRASBOURG

PRÉSENTATION DES TRAVAUX

pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES

par

Éric Violard

Construction de programmes parallèles par des techniques de transformations

Membres du jury :

Pascal Gribomont	Rapporteurs
Gaétan Hains	
Jean-François Dufourd	
Dominique Mery	Examinateurs
Guy-René Perrin	

Table des matières

Table des figures

Introduction

Les architectures parallèles représentent une puissance de calcul toujours plus grande (avec l'émergence des grilles de calcul, les ressources de calcul peuvent être encore augmentées). Ces architectures permettent d'envisager la résolution de problèmes toujours plus complexes représentant des volumes de données et de calculs considérables. Cependant, un grand défi est de programmer ces architectures de façon à exploiter efficacement leur puissance de calcul.

La programmation parallèle relie deux objets : un problème dont la résolution nécessite un certain nombre de calculs et une architecture parallèle. Le langage parallèle est l'intermédaire utilisé par le programmeur et le compilateur pour relier ces deux objets. Cette relation est une affaire de capacité de programmation et de techniques de compilation. Elle peut être plus ou moins facilement établie selon le modèle de programmation sous-jacent au langage. Si ce modèle n'est pas suffisamment abstrait, alors le programmeur aura beaucoup de difficultés à apporter ses connaissances du problème, s'il est trop abstrait alors il sera difficile pour le compilateur d'atteindre une implantation efficace sur l'architecture cible. En particulier, parmi les différents modèles de programmation qui ont été successivement proposés, le data-parallélisme est d'un grand intérêt pour concilier haut niveau de programmation et efficacité des implantations.

Nous pensons que la capacité de maîtriser l'efficacité des calculs peut être obtenue en développant des transformations formelles d'énoncés qui s'appliquent étape par étape et mènent avec sûreté d'un énoncé de problème à un énoncé correspondant à une implantation efficace sur une architecture cible. Ces transformations sont les éléments constitutifs d'un raisonnement permettant de concevoir des programmes parallèles ou d'obtenir des implantations efficaces. Nos travaux concernent la définition de ces transformations et portent sur ces questions de méthodologie et de raisonnement pour construire les programmes parallèles et leurs implantations.

A notre sens, les langages parallèles devraient être un support de ces transformations afin que le raisonnement puisse être conduit par le compilateur ou/et par le programmeur. Tout modèle de programmation établit un médium permettant au programmeur et au compilateur d'échanger leurs connaissances respectives – dans le domaine du problème pour le programmeur, dans le domaine de l'architecture pour le compilateur – afin d'obtenir une implantation efficace. A notre avis, les modèles de programmation devraient établir un haut degré de coopération entre programmeur et compilateur. Nous pensons que cette approche «par transformation d'énoncés» est une bonne approche pour définir des modèles de programmation et des langages parallèles qui répondent aux besoins des utilisateurs.

La présentation de nos travaux est organisée comme suit : le premier chapitre présente nos

développements autour du cadre formel PEI proposé dans ma thèse de doctorat et conçu pour raisonner sur les programmes parallèles et leurs implantations. Le deuxième chapitre présente nos travaux relatifs au data-parallélisme et à sa définition : nous avons proposé un domaine sémantique pour les langages data-parallèles qui est issu de notre cadre formel. Le troisième et dernier chapitre présente des applications de nos travaux théoriques.

Chapitre 1

Un cadre formel pour raisonner

1.1 Introduction

Ma thèse de doctorat [63] a proposé un cadre formel, nommé PEI, pour unifier deux approches classiques et complémentaires pour construire un programme parallèle par des techniques de transformations :

- le raffinement étape par étape d'une spécification jusqu'à obtenir une spécification exécutable selon un certain modèle de calcul [6, 5, 51] et
- la synthèse d'un programme à partir d'un système d'équations récurrentes [42, 54, 47].

Ces approches sont associées à des formalismes spécifiques pour supporter les transformations.

Le raffinement de spécifications adresse un éventail très large de problèmes et de modèles de calcul. Il est appliqué dans divers travaux [38, 43] à la construction de programmes parallèles. Les formalismes de dérivation (LINDA [32], UNITY [12], GAMMA [21], etc.) supportent un calcul de raffinement et les spécifications sont exprimées par des prédicats. Avec cette approche, on vise surtout la correction des programmes, mais sans traiter véritablement de leur efficacité.

L'autre approche est utilisée en parallélisation automatique. Il a été montré dans [42] qu'un système d'équations récurrentes permet de formaliser des problèmes «systolisables» en introduisant les notions de cadencement et d'allocation des calculs. Ces idées ont été largement développées dans la littérature qui concerne la synthèse de réseaux systoliques ou la parallélisation de nids de boucles [22, 48, 17]. Des formalismes ad'hoc (ALPHA [50], CRYSTAL [15], LACS [55], etc.) permettent l'expression des équations et de leurs transformations. Avec cette approche, on vise évidemment les critères d'efficacité, mais en adressant une classe relativement restreinte de problèmes.

Le cadre formel PEI [63, 69, 65] bénéficie des avantages de ces deux approches. Ce chapitre présente ce cadre formel sur lequel s'appuie une grande partie de mes recherches et fait la synthèse des travaux que nous avons menés pour mettre en évidence son pouvoir de dérivation.

1.2 Définition de PEI

Le cadre formel PEI repose sur certaines notions mathématiques, peu nombreuses mais puissantes. En premier lieu, la notion de multi-ensemble de valeurs. Pour que ces valeurs puissent être accédées, elles sont placées sur des domaines discrets et forment des objets PEI.

D'un point de vue opérationnel, de tels domaines peuvent abstraire le placement des calculs sur un domaine espace-temps, alors que du point de vue d'une spécification, ils peuvent exprimer la géométrie naturelle des structures de données, telles que les tableaux, par exemple. L'ensemble des objets PEI est muni de trois opérations externes qui, soit calculent les valeurs des objets, soit expriment des dépendances de données, soit changent les domaines discrets des objets. Ces opérations sont les briques de base d'un calcul de raffinement permettant la dérivation ou la transformation de programmes. En PEI, spécifications et programmes sont des systèmes d'équations non-orientées (qui connectent deux objets PEI), appelés énoncés. Ces énoncés généralisent les systèmes d'équations récurrentes.

1.2.1 Objets - Opérations

De manière générale, nous pouvons considérer qu'un problème est une relation entre des *multi-ensembles* de valeurs en entrée et en sortie. Bien sûr, programmer implique de «placer» ces valeurs d'une certaine façon pour les organiser et les traiter. En calcul scientifique, par exemple, on utilise des tableaux qui sont des fonctions sur des indices : l'ensemble d'indices, qui sert de domaine de référence, est une partie de $\mathbb{Z}^n$. Un objet PEI est un tel multi-ensemble de valeurs placées sur un domaine de référence discret. Étant donné un objet PEI, on peut obtenir un autre objet PEI qui représente le même multi-ensemble en changeant de domaine de référence. Deux tels objets sont dits *équivalents*.

L'ensemble des objets PEI est muni de quatre opérations. Les trois premières opérations (externes) modifient un objet PEI en utilisant une fonction partielle. La dernière opération (interne) combine plusieurs objets PEI pour en former un seul.

le *réindiçage* (ou changement de base) permet de changer de domaine de référence.

Le réindiçage est déterminé par une bijection h entre indices : le nouveau domaine de référence est l'image par h de l'ancien.

le *déplacement* (ou opération géométrique ou encore routage) déplace les valeurs sur le domaine de référence.

Le déplacement est déterminé par une fonction g entre indices : la valeur placée à l'indice z dans le nouvel objet est celle qui était placée à l'indice $g(z)$ dans l'ancien.

le *calcul* (ou opération fonctionnelle) applique une même fonction f sur chaque valeur.

la *superposition* forme des séquences de valeurs en prenant les valeurs des arguments.

La valeur placée à un certain indice dans le résultat est la séquence des valeurs placées au même indice dans les arguments.

1.2.2 Formalisme - Énoncés

Nous avons défini un formalisme minimal pour supporter le raisonnement formel. Le formalisme inclue une notation pour les opérations et les fonctions partielles qu'elles utilisent. Cette notation permet de construire des expressions d'objets PEI . Ces expressions sont construites à partir de *constantes*, de variables et des notations d'opération.

- Notation des constantes : Un objet PEI est *constant* ssi toutes ses valeurs sont égales. Pour tout élément c d'un type de valeurs, c note un objet PEI constant de valeurs égales à c.

- Notation des opérations :

$h :: X$	note l'application du réindiçage à un objet X et une fonction h,
$X \lhd g$	note l'application du déplacement à un objet X et une fonction g,
$f \rhd X$	note l'application du calcul global à un objet X et une fonction f,
$X \ /;/ \ Y$	note l'application de la superposition à deux objets X et Y.

- Notation des fonctions partielles : Elle est empruntée du lambda-calcul. la notation pour une fonction partielle f de domaine de définition $\{ x \mid P(x) \}$ est $\lambda x \mid (P(x))\ .f(x)$. Pour une fonction totale, la notation est la même qu'en lambda-calcul. De plus, nous notons $f \circ g$, la composée de deux fonctions et $f \# g$, une fonction définie sur deux sous-domaines disjoints. Enfin, $inv(f)$ note l'inverse d'une fonction injective notée f.

Le formalisme permet d'écrire des *énoncés* qui sont des ensembles d'équations non-orientées avec des variables identifiées en tant que variable d'entrée et de sortie (les autres variables sont appelées variables intermédiaires). Chaque équation connecte deux expressions d'objets PEI . Les définitions des fonctions partielles sont reportées dans le contexte de l'énoncé et les paramètres indiqués entre crochets juste après le nom de l'énoncé.

Un énoncé exprime une relation entre des objets PEI en entrée et en sortie.

Exemple 1 (Équation de la chaleur)

$$\mathsf{DiffHeat}[n, T, r] : V \to U$$

$$\{$$
$$\mathsf{rod} :: V0 = V$$
$$U \lhd \mathsf{bord} = 0$$
$$U \lhd \mathsf{thinrod} = V0$$
$$U \lhd \mathsf{inside} = \mathsf{approx} \rhd (U \lhd \mathsf{left} \ /;/ \ U \lhd \mathsf{middle} \ /;/ \ U \lhd \mathsf{right})$$
$$\}$$

$$\begin{array}{lcl}
\mathsf{rod}(i,j) & = & (1 \le i \le n \wedge j = 0)\ .(i) \\
\mathsf{bord}(i,j) & = & ((i = 1 \vee i = n) \wedge 0 < j \le T)\ .(i,j) \\
\mathsf{thinrod}(i,j) & = & (1 \le i \le n \wedge j = 0)\ .(i,j) \\
\mathsf{inside}(i,j) & = & (1 < i < n \wedge 0 < j \le T)\ .(i,j) \\
\mathsf{left}(i,j) & = & (1 < i < n \wedge 0 < j \le T)\ .(i{+}1, j{-}1) \\
\mathsf{middle}(i,j) & = & (1 < i < n \wedge 0 < j \le T)\ .(i, j{-}1) \\
\mathsf{right}(i,j) & = & (1 < i < n \wedge 0 < j \le T)\ .(i{-}1, j{-}1) \\
\mathsf{approx}(a;b;c) & = & r.a + (1 - 2r).b + r.c
\end{array}$$

Cet énoncé en PEI spécifie le problème de la diffusion de la chaleur dans une fine tige de métal. Il est construit de la manière suivante :

Considérons une fine tige de métal dont les extrémités sont maintenues à $0\,^\circ$. La température $U(x,t)$ de la tige à l'instant t et à la distance x d'une de ces extrémités, vérifie l'équation différentielle suivante :

$$\frac{\partial U}{\partial t} = \frac{\partial U}{\partial x^2} \tag{1.1}$$

De façon à simuler la diffusion de la chaleur dans la tige, une solution numérique est obtenue en discrétisant l'équation dans le temps et l'espace [58]. Une approximation en différences finies de l'équation 1.1 est :

$$U_{i,j} = rU_{i-1,j-1} + (1-2r)U_{i,j-1} + rU_{i+1,j-1} \tag{1.2}$$

où r est une constante qui ne dépend que des paramètres de discrétisation.

En Pei, U s'exprime par un objet Pei que nous choisissons de noter U. Le domaine de référence de U est un domaine de $\mathbb{Z}^2$ et les valeurs de U sont placées dans le rectangle $[1..n] \times [0..T]$ où n et T sont les limites discrètes de l'espace et le temps discrétisés.

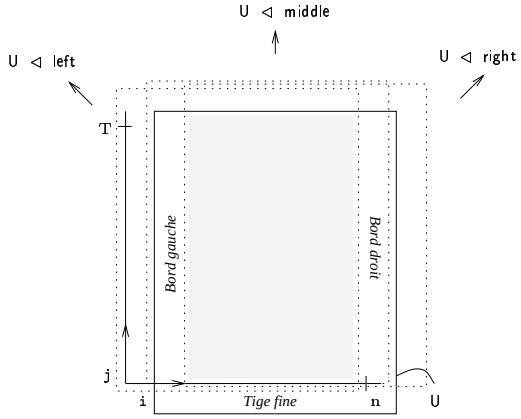


Fig. 1.1 – *Construction par superposition.*

Les dépendances de l'équation 1.2 s'expriment en Pei par des déplacements. Ces déplacements définissent quatre sous-domaines dans le domaine des valeurs de U (voir la figure 1.1) : les points de la tige de métal (tels que j=0), les bords gauche et droit et le domaine intérieur (en grisé). Les valeurs aux bords sont connues : la température initiale en chaque point de la tige de métal et 0 sur les bords gauche et droit. Les valeurs à l'intérieur du rectangle sont calculées à partir des valeurs des objets notés (U ◁ left), (U ◁ middle) et (U ◁ right) qui sont obtenus en décalant U respectivement en haut à gauche, en haut et en haut à droite (selon l'orientation de la figure) : les valeurs de ces objets sont regroupées par superposition et combinées par un calcul pour former les valeurs de U à l'intérieur du rectangle, comme l'exprime cette équation :

$$\mathsf{U} \lhd \mathsf{inside} = \mathsf{approx} \rhd (\mathsf{U} \lhd \mathsf{left} \;/;/\; \mathsf{U} \lhd \mathsf{middle} \;/;/\; \mathsf{U} \lhd \mathsf{right})$$

avec :
$$\begin{array}{lcl} \mathsf{inside(i,j)} & = & (1 < \mathsf{i} < \mathsf{n} \wedge 0 < \mathsf{j} \leq \mathsf{T})\,.(\mathsf{i},\mathsf{j}) \\ \mathsf{left(i,j)} & = & (1 < \mathsf{i} < \mathsf{n} \wedge 0 < \mathsf{j} \leq \mathsf{T})\,.(\mathsf{i}{+}1,\mathsf{j}{-}1) \\ \mathsf{middle(i,j)} & = & (1 < \mathsf{i} < \mathsf{n} \wedge 0 < \mathsf{j} \leq \mathsf{T})\,.(\mathsf{i},\mathsf{j}{-}1) \\ \mathsf{right(i,j)} & = & (1 < \mathsf{i} < \mathsf{n} \wedge 0 < \mathsf{j} \leq \mathsf{T})\,.(\mathsf{i}{-}1,\mathsf{j}{-}1) \end{array}$$

et : $\mathsf{approx(a;b;c)} = \mathsf{r.a} + (1-2\mathsf{r}).\mathsf{b} + \mathsf{r.c}$.

Considérons maintenant le vecteur dont les éléments sont les températures initiales de la tige de métal. Ce vecteur est naturellement associé à un objet PEI, noté V, placé sur la ligne $[1..n]$ de $\mathbb{Z}$. En PEI, la relation entre U et V peut s'écrire comme suit (en utilisant une variable intermédiaire V0) :

$$\mathsf{rod} :: \mathsf{V0} = \mathsf{V}$$
$$\mathsf{U} \lhd \mathsf{thinrod} = \mathsf{V0}$$

avec :

$$\begin{array}{lcl} \mathsf{rod(i,j)} & = & (1 \leq \mathsf{i} \leq \mathsf{n} \wedge \mathsf{j} = 0)\ .(\mathsf{i}) \\ \mathsf{thinrod(i,j)} & = & (1 \leq \mathsf{i} \leq \mathsf{n} \wedge \mathsf{j} = 0)\ .(\mathsf{i},\mathsf{j}) \end{array}$$

La première équation exprime la relation entre V et V0 : les valeurs de V0 sont les mêmes que celles de V mais placées sur une ligne de $\mathbb{Z}^2$. Cette relation s'exprime par un réindiçage. La deuxième équation exprime que les valeurs de V0 sont celles de U sur le domaine des points de la tige de métal.

En regroupant les équations, nous obtenons l'énoncé précédent.
◇

1.2.3 Définition mathématique - Sémantique d'un énoncé

Soit V, un ensemble de valeurs. Nous appelons *placement* d'un multi-ensemble M de valeurs dans V, une fonction partielle $v : \mathbb{Z}^n \rightharpoonup V$ telle que $M = \prec v(z) \mid z \in \mathrm{def}(v) \succ$. Nous appelons *domaine des valeurs*, le domaine de définition $\mathrm{def}(v)$,

Deux placements quelconques d'un même multi-ensemble diffèrent d'une bijection (d'un domaine de valeurs vers un autre). Cela signifie que tout choix de placement définissant un objet PEI est arbitraire et diffère d'une bijection par rapport à ce qu'on pourrait considérer comme un *placement de référence*. En intégrant cette bijection à la définition des objets PEI, on peut définir correctement la relation entre deux objets PEI qui représentent le même multi-ensemble : nous dirons que leur placement de référence est le même.

Un objet PEI est donc un couple $(v : \sigma)$ formé d'un placement v et d'une bijection σ qui permet de retrouver le placement de référence à partir du placement v. Nous appelons *domaine de référence*, le domaine de définition $\mathrm{def}(\sigma)$ et *placement de référence*, le placement défini par $v \circ \sigma^{-1}$ qui est valide (i.e. place toutes les valeurs) seulement si le domaine de valeur est inclus dans le domaine de référence, i.e. $\mathrm{def}(v) \subseteq \mathrm{def}(\sigma)$.

Définition 1 (Objet PEI) Soit V, un ensemble de valeurs. Un *objet* PEI à valeurs dans V est un couple $(v : \sigma)$ formé d'une fonction partielle $v : \mathbb{Z}^n \rightharpoonup V$ et d'une bijection $\sigma : \mathbb{Z}^n \rightharpoonup \mathbb{Z}^m$ telle que $\mathrm{def}(v) \subseteq \mathrm{def}(\sigma)$.

Comme nous l'avons mentionné en section 1.2.1, deux objets PEI sont considérés comme *équivalents* s'ils représentent le même multi-ensemble. Nous considérons que deux objets PEI peuvent être équivalents à deux niveaux. Au premier niveau, appelé *équivalence faible*, les objets ont les mêmes valeurs indépendamment du placement de référence : qu'ils représentent

ou non le même multi-ensemble. Au second niveau, appelé *équivalence forte*, les objets ont les mêmes valeurs et le même placement de référence : ils représentent le même multi-ensemble. Voici la définition de l'équivalence des objets PEI :

Définition 2 (Équivalence faible) Soient X et Y, deux objets PEI. X et Y sont dits *faiblement équivalents*, si et seulement si $\prec v_X(z) \mid z \in \mathrm{def}(v_X) \succ = \prec v_Y(z) \mid z \in \mathrm{def}(v_Y) \succ$.

Définition 3 (Équivalence forte) Soient X et Y, deux objets PEI. X et Y sont dits *fortement équivalents*, si et seulement si $v_X \circ \sigma_X^{-1} = v_Y \circ \sigma_Y^{-1}$.

La définition des objets PEI conduit à ces définitions des opérations :

Définition 4 (Réindiçage) Soit $X = (v : \sigma)$, un objet PEI. Soit h, une bijection : $\mathbb{Z}^n \rightharpoonup \mathbb{Z}^m$. telle que $\mathrm{def}(v) \subseteq \mathrm{def}(h)$. Le réindiçage est l'opération qui, appliquée à X et h, résulte en l'objet PEI $(v \circ h^{-1} : \sigma \circ h^{-1})$.

Définition 5 (Déplacement) Soit $X = (v : \sigma)$, un objet PEI. Soit g, une fonction : $\mathbb{Z}^n \rightharpoonup \mathbb{Z}^n$ telle que $\mathrm{def}(g) \subseteq \mathrm{def}(\sigma)$ et $\mathrm{img}(g) \subseteq \mathrm{def}(v)$. Le déplacement est l'opération qui, appliquée à X et g, résulte en l'objet PEI $(v \circ g : \sigma)$.

Définition 6 (Calcul) Soient V et W, deux ensembles de valeurs. Soit $X = (v : \sigma)$, un objet PEI à valeurs dans V. Soit f une fonction partielle : $V \rightharpoonup W$. Le calcul est l'opération qui, appliquée à X et f, résulte en l'objet PEI $(f \circ v : \sigma)$.

Dans la définition suivante, $seq[V]$ est l'ensemble des séquences non vides de valeurs dans V et $a; b$ est la concaténation de des séquences a et b. Toute valeur scalaire est considérée comme une séquence à un élément.

Définition 7 (Superposition) Soit V, un type de valeurs. Soit $X = (v : \sigma)$ et $Y = (w : \sigma)$, deux objets PEI à valeurs dans le même ensemble $seq[V]$. L'opération de superposition est l'opération qui, appliquée à X et Y résulte en l'objet PEI $(x : \sigma)$, où x est la fonction partielle de même domaine et codomaine que v et w, définie par :

$$x(z) = \begin{cases} v(z) & \text{if } x \in \mathrm{def}(v) \setminus \mathrm{def}(w) \\ v(z); w(z) & \text{if } x \in \mathrm{def}(v) \cap \mathrm{def}(w) \\ w(z) & \text{if } x \in \mathrm{def}(w) \setminus \mathrm{def}(v). \end{cases}$$

La sémantique d'un énoncé est induite de la sémantique d'une équation. La sémantique d'une équation (ex : $\mathsf{Y} = \mathsf{X} \lhd \mathsf{g}$) est un prédicat : ce prédicat est obtenu en faisant la conjonction de : *(i)* toutes les conditions à vérifier pour appliquer les opérations apparaissant aux deux membres de l'équation (ex : $\mathrm{def}(\mathsf{g}) \subseteq \mathrm{def}(\sigma_{\mathsf{X}}) \wedge \mathrm{img}(\mathsf{g}) \subseteq \mathrm{def}(v_{\mathsf{X}})$), et *(ii)* l'égalité entre les fonctions v et σ des deux objets PEI qui sont les résultats des opérations aux deux membres de l'équation (ex : $v_{\mathsf{Y}} = v_{\mathsf{X}} \circ \mathsf{g} \wedge \sigma_{\mathsf{Y}} = \sigma_{\mathsf{X}}$). Si ce prédicat est vrai, alors on dit que les objets PEI notés par les variables liées dans cette équation vérifient l'équation.

La sémantique d'un énoncé est la relation entre les objets PEI qui, lorsqu'ils sont notés par les variables en entrée et en sortie, vérifient toutes les équations.

1.2.4 Raffinement d'énoncés en PEI

Le raffinement en PEI est de deux types :

- Le raffinement, dit *dénotationnel*, qui consiste à restreindre la relation entre objets PEI pour obtenir une fonction. Il fait décroître le nombre de solution du système d'équation, jusqu'à obtenir une solution unique : alors, l'énoncé devient un programme.
- Le raffinement, dit *opérationnel*, qui consiste à définir un certain ordre opérationnel, au sens suivant :

  Définition 8 (Ordre opérationnel) Soit $\ll$, un ordre partiel sur $\mathbb{Z}^m$ et $(v : \sigma)$, un objet PEI où $\sigma : \mathbb{Z}^n \rightharpoonup \mathbb{Z}^m$. La relation, notée $\vdash$ définie par :

  $$\forall z, z' \in \text{def}(v) \quad (v(z) \vdash v(z') \text{ ssi } \sigma(z) \ll \sigma(z'))$$

  est un ordre partiel sur def(v), appelé ordre opérationnel.

  Le choix de l'ordre partiel $\ll$ prédétermine la sémantique opérationnelle d'un programme. Considérons par exemple une bijection σ telle que $\sigma(z) = (p(z), t(z))$, où $p : \mathbb{Z}^n \rightharpoonup \mathbb{Z}^{m-1}$ et $t : \mathbb{Z}^n \rightharpoonup \mathbb{N}$. Une telle définition est une façon classique de définir un cadencement et une allocation des calculs sur un ensemble de processeurs virtuels.
  - si $\ll$ est tel que $\sigma(z) \ll \sigma(z') \Leftrightarrow t(z) < t(z')$, alors la sémantique opérationnelle induite définit seulement le cadencement des calculs,
  - si $\ll$ est tel que $\sigma(z) \ll \sigma(z') \Leftrightarrow p(z) = p(z') \ \wedge \ t(z) < t(z')$, alors la sémantique opérationnelle induite définit l'allocation sur les processeurs en plus du cadencement.

  Ce type de raffinement consiste à expliciter une bijection σ qui introduit un ordre opérationnel approprié. Il est fondé sur l'équivalence des objets PEI et les transformations pour raffiner opérationnellement un énoncé reposent sur le réindiçage.

Définition du raffinement

La définition du raffinement en PEI [33, 72] est proche de la définition de Knapp [43] pour Unity [12], mais en restant dans la logique du premier ordre.

Nous associons à un énoncé P, une propriété $sat(P)$ dont nous disons qu'elle est satisfaite par P. Pour définir cette propriété, nous distinguons parmi les équations de P, les pré-équations Pre qui lient uniquement des variables en entrée, et les post-équations Post qui sont les autres équations de P. La propriété de satisfaction dit que pour tous les objets PEI en entrée qui vérifient les pré-équations, la solution est formée des objets PEI en sortie qui vérifient les post-équations, ce que nous notons $sat(P) : \mathsf{Pre} \implies \mathsf{Post}$.

Définition 9 (Raffinement d'énoncés) Soient P et P', deux énoncés PEI. Nous disons que P est raffiné par P', et nous notons $P \sqsubseteq P'$, si $sat(P') \implies sat(P)$.

Cette définition du raffinement permet d'appliquer les procédés classiques de raffinement comme l'affaiblissement d'une pré-équation ou le renforcement d'une post-équation. Des procédés plus spécifiques, également applicables en PEI, sont l'ajout ou la suppression d'une variable intermédiaire.

Une façon plus générale de raffiner un énoncé PEI consiste à substituer un objet équivalent à un objet donné, au sens de l'équivalence forte ou faible. Cela peut être réalisé en généralisant la

définition précédente pour prendre en compte l'équivalence et en introduisant une implication modulo l'équivalence, notée $\overset{\equiv}{\Longrightarrow}$ et formellement définie dans [33, 72] :

Définition 10 (Raffinement d'énoncés) Soient P et P', deux énoncés PEI. Nous disons que P est raffiné par P', et nous notons $P \sqsubseteq P'$, ssi $sat(P') \overset{\equiv}{\Longrightarrow} sat(P)$.

Cette définition du raffinement permet notamment d'établir la propriété suivante utilisée pour raffiner opérationnellement un énoncé :

Propriété 1 Soient P, un énoncé PEI, X, une variable de P, et h, une bijection : $\mathbb{Z}^n \rightharpoonup \mathbb{Z}^m$ notée h. Si le prédicat $\mathrm{def}(\sigma_{\mathsf{X}}) \subseteq \mathrm{img}(\mathsf{h})$ peut être déduit de P, alors P est raffiné par l'énoncé obtenu à partir de P en remplaçant toutes les occurrences de X par $\mathsf{h} :: \mathsf{X}$.

Calcul de raffinement

La définition du raffinement induit un calcul de raffinement formé d'un ensemble de règles pour raffiner un énoncé. Les règles que nous avons élaboré sont données dans [72] et leur preuve peut être trouvée dans [33].

Certaines règles dites algébriques correspondent à des propriétés algébriques des opérations sur les objets PEI. Elles permettent de simplifier ou plus généralement réécrire une expression d'objets PEI. Une telle règle s'écrit $\mathsf{E} \sqsubseteq \mathsf{E}'$, où E et E' sont des expressions d'objets PEI, et signifie que l'on obtient un énoncé raffiné en substituant E' à E dans une post-équation ou en substituant E à E' dans une pré-équation. Voici quelques exemples de règles algébriques :

$$(\mathsf{h} :: \mathsf{X}) \ /;/ \ (\mathsf{h} :: \mathsf{Y}) \ \sqsubseteq \ \mathsf{h} :: (\mathsf{X} \ /;/ \ \mathsf{Y}) \tag{1.3}$$

$$\mathsf{f} \rhd (\mathsf{h} :: \mathsf{X}) \ \sqsubseteq \ \mathsf{h} :: (\mathsf{f} \rhd \mathsf{X}) \tag{1.4}$$

$$\mathsf{h} :: (\mathsf{h}' :: \mathsf{X}) \ \sqsubseteq \ (\mathsf{h} \circ \mathsf{h}') :: \mathsf{X} \tag{1.5}$$

$$\mathsf{X} \lhd (\mathsf{g} \circ \mathsf{g}') \ \sqsubseteq \ (\mathsf{X} \lhd \mathsf{g}) \lhd \mathsf{g}' \tag{1.6}$$

Certaines règles algébriques sont assorties d'une condition d'application et s'écrivent : $\mathsf{E} \sqsubseteq \mathsf{E}'$ avec C. La condition C est un prédicat qui porte sur les domaines des fonctions partielles ou les domaines des valeurs et de référence des variables. En voici un exemple :

$$\begin{aligned}(\mathsf{h} :: \mathsf{X}) \lhd \mathsf{g} \ \sqsubseteq \ & \mathsf{h} :: (\mathsf{X} \lhd (\mathsf{inv}(\mathsf{h}) \circ \mathsf{g} \circ \mathsf{h})) \\ & \text{avec } \ \mathrm{def}(\mathsf{g}) \cup \mathrm{img}(\mathsf{g}) \subseteq \mathrm{img}(\mathsf{h}) \wedge \mathrm{def}(v_{\mathsf{X}}) \subseteq \mathrm{def}(\mathsf{h})\end{aligned} \tag{1.7}$$

La règle peut s'appliquer uniquement si la condition peut être déduite de l'énoncé obtenu après application de la règle.

D'autres règles permettent de réécrire en même temps les deux membres d'une équation. Leur notation et sémantique sont similaires aux règles algébriques. Par exemple :

$$\begin{aligned}\mathsf{X} = \mathsf{Y} \ \sqsubseteq \ & \mathsf{h} :: \mathsf{X} = \mathsf{h} :: \mathsf{Y} \\ & \text{avec } \ \mathrm{def}(\sigma_{\mathsf{X}}) \cup \mathrm{def}(\sigma_{\mathsf{Y}}) \subseteq \mathrm{def}(\mathsf{h})\end{aligned} \tag{1.8}$$

$$\begin{aligned}\mathsf{h} :: \mathsf{X} = \mathsf{h} :: \mathsf{Y} \ \sqsubseteq \ & \mathsf{X} = \mathsf{Y} \\ & \text{avec } \ \mathrm{def}(v_{\mathsf{X}}) \cap \mathrm{def}(v_{\mathsf{Y}}) \subseteq \mathrm{def}(\mathsf{h})\end{aligned} \tag{1.9}$$

Nous avons défini un procédé de déduction permettant, étant donné un énoncé en PEI, de vérifier les conditions d'application des règles de raffinement. Ce procédé consiste en un système d'inférence de type où le type d'un objet PEI $(v : \sigma)$ est le couple $(\text{def}(v), \text{def}(\sigma))$ formé de son domaine de valeur et de son domaine de référence. L'énoncé est bien typé si et seulement si, pour toute expression, disons X, d'objet PEI, le domaine de valeur est inclus dans le domaine de référence, i.e. $\text{def}(v_{\mathsf{X}}) \subseteq \text{def}(\sigma_{\mathsf{X}})$, conformément à la définition d'un objet PEI. Ce procédé est décrit dans [66].

1.3 Raisonnement en PEI

Nous présentons dans cette section les travaux visant à montrer le pouvoir de dérivation du cadre formel PEI et comment il permet d'adapter progressivement un énoncé à des contraintes d'ordre opérationnel.

1.3.1 Implantation efficace sur une architecture cible

Nous avons montré que les transformation en PEI généralisent les transformations en parallélisation automatique [65, 64]. Les techniques de transformations en parallélisation automatique reposent sur l'application d'un *changement de base* pour exprimer, dans une base espace-temps, le domaine des points de calcul parcourus par les indices d'une variable d'un système d'équations recurrentes. Une fois ce changement de base effectué, le domaine décrit une exécution parallèle [54, 18, 48].

Le changement de base est une transformation intrinsèque en PEI puisque les objets PEI intègrent une bijection d'un placement arbitraire vers le placement de référence.

En PEI, un changement de base est défini par un réindiçage et son application sur une variable, disons X, d'un énoncé, consiste à utiliser la propriété (1) pour substituer (h :: X) à X dans l'énoncé, factoriser le réindiçage en utilisant un sous-ensemble confluent de règles algébriques (notamment la règle [1.7]) et utiliser la règle [1.9] pour obtenir une nouvelle définition de la variable.

Un changement de base en parallélisation automatique est un cas particulier de changement de base en PEI lorsque l'énoncé en PEI exprime un système d'équations récurrentes. Alors, les résultats en parallélisation automatique peuvent être utilisés : l'analyse de dépendances permet de déterminer un réindiçage défini par une bijection composée d'une fonction de cadencement et d'une fonction d'allocation des calculs. Ce réindiçage définit un changement de base en PEI dont l'application mène à un énoncé correspondant à une implantation parallèle efficace.

Exemple 2 (Transformation espace×temps d'un nid de boucles)

La transformation de nids de boucles est une technique classique en parallélisation automatique (voir la littérature étendue sur ce sujet [27, 7, 47], etc.). Considérons le nid de boucles suivant où $a_{i,1}$, $i \in [1..n]$ et $a_{1,j}$, $j \in [1..n]$ sont les données en entrée :

```
do i=2,n
 do j=2,n
  a(i,j)=a(i-1,j)+a(i,j-1)
```

```
 enddo
enddo
```

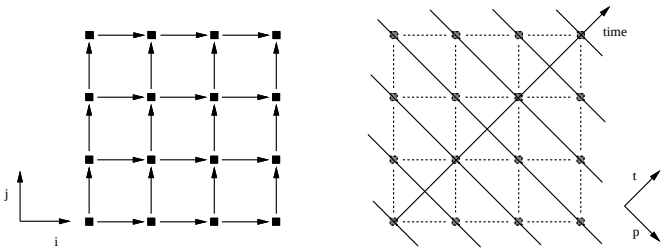


FIG. 1.2 – *Dépendances et transformation espace×temps.*

A partir d'une analyse de dépendances (voir la figure 1.2), une transformation espace×temps affine peut s'appliquer, définie par :

$$\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix}$$

Le nouvel espace d'itération peut être parcouru par un front de calcul, à l'intérieur d'une boucle séquentielle sur le temps. Une telle technique de compilation correspond à un changement de base en PEI. Voici le même exemple écrit en utilisant le formalisme PEI :

$\mathsf{NestedLoop[n]} : \mathsf{D} \to \mathsf{A}$

$\{$
$\mathsf{bord} :: \mathsf{D0} = \mathsf{D}$
$\mathsf{A} \lhd \mathsf{bord} = \mathsf{D0}$
$\mathsf{A} \lhd \mathsf{next} = \mathsf{add} \rhd (\mathsf{A} \lhd \mathsf{right}\ /;/\ \mathsf{A} \lhd \mathsf{up})$
$\}$

$\mathsf{bord(i,j)} = (1 \le \mathsf{i,j} \le \mathsf{n} \wedge (\mathsf{i} = 1 \vee \mathsf{j} = 1))\ .(\mathsf{i,j})$
$\mathsf{next(i,j)} = (1 < \mathsf{i,j} \le \mathsf{n})\ .(\mathsf{i,j})$
$\mathsf{add(a;b)} = (\mathsf{a+b})$
$\mathsf{right(i,j)} = (1 < \mathsf{i,j} \le \mathsf{n})\ .(\mathsf{i-1,j})$
$\mathsf{up(i,j)} = (1 < \mathsf{i,j} \le \mathsf{n})\ .(\mathsf{i,j-1})$

Considérons la bijection suivante pour le réindiçage :

$\mathsf{time\_space(t,p)} = ((\mathsf{p+t-n+1})\%2 = 0)\ .((\mathsf{p+t-n+1})/2, (\mathsf{t-p+n+1})/2)$

D'après la définition du réindiçage en PEI, cette bijection est l'inverse de la bijection qui définit la transformation espace×temps mentionnée ci-dessus. Les conditions d'application de la propriété (1) sont satisfaites et l'application de cette propriété mène à ces équations :

$(\mathsf{time\_space} :: \mathsf{A}) \lhd \mathsf{bord} = \mathsf{D0}$
$(\mathsf{time\_space} :: \mathsf{A}) \lhd \mathsf{next} = \mathsf{add} \rhd ((\mathsf{time\_space} :: \mathsf{A}) \lhd \mathsf{right}\ /;/\ (\mathsf{time\_space} :: \mathsf{A}) \lhd \mathsf{up})$

En factorisant le réindiçage (en appliquant la règle [1.7]), les équations se réécrivent :

$$\mathsf{time\_space} :: (\mathsf{A} \lhd \mathsf{bord}') = \mathsf{D0}$$
$$\mathsf{time\_space} :: (\mathsf{A} \lhd \mathsf{next}') = \mathsf{time\_space} :: (\mathsf{add} \rhd (\mathsf{A} \lhd \mathsf{right}' \ /;/ \ \mathsf{A} \lhd \mathsf{up}'))$$

où bord', right' et up' sont obtenues par la transformation.
Une expression telle que $(\mathsf{time\_space} :: \mathsf{A}) \lhd \mathsf{right}$ est transformée en appliquant les règles algébriques. Elle se réécrit $\mathsf{time\_space} :: (\mathsf{A} \lhd \mathsf{right}')$ avec :

$$\mathsf{right}' = \mathsf{inv}(\mathsf{time\_space}) \circ \mathsf{right} \circ \mathsf{time\_space}$$

En procédant de façon similaire pour les autres fonctions, nous obtenons :

$$\begin{array}{lcl}
\mathsf{bord}'(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{p}{+}\mathsf{t}{-}\mathsf{n}, \mathsf{t}{-}\mathsf{p}{+}\mathsf{n} \le \mathsf{n} \wedge (\mathsf{p} = \mathsf{n}{-}\mathsf{t}{+}1 \vee \mathsf{p} = \mathsf{t}{+}\mathsf{n}{-}1)) \ .(\mathsf{t},\mathsf{p}) \\
\mathsf{next}'(\mathsf{t},\mathsf{p}) & = & (1 < \mathsf{p}{+}\mathsf{t}{-}\mathsf{n}, \mathsf{t}{-}\mathsf{p}{+}\mathsf{n} \le 2\mathsf{n}{-}1) \ .(\mathsf{t},\mathsf{p}) \\
\mathsf{right}'(\mathsf{t},\mathsf{p}) & = & (1 < \mathsf{p}{+}\mathsf{t}{-}\mathsf{n}, \mathsf{t}{-}\mathsf{p}{+}\mathsf{n} \le 2\mathsf{n}{-}1 \wedge (\mathsf{p}{+}\mathsf{t}{-}\mathsf{n}{+}1)\%2 = 0)) \ .(\mathsf{t}{-}1, \mathsf{p}{-}1) \\
\mathsf{up}'(\mathsf{t},\mathsf{p}) & = & (1 < \mathsf{p}{+}\mathsf{t}{-}\mathsf{n}, \mathsf{t}{-}\mathsf{p}{+}\mathsf{n} \le 2\mathsf{n}{-}1 \wedge (\mathsf{p}{+}\mathsf{t}{-}\mathsf{n}{+}1)\%2 = 0)) \ .(\mathsf{t}{-}1, \mathsf{p}{+}1)
\end{array}$$

En utilisant la règle [1.9] pour simplifier la dernière équation et l'élimination de Fourier pour simplifier les prédicats des définitions de fonctions, nous obtenons le programme PEI suivant :

$$\mathsf{NestedLoop[n]} : \mathsf{D} \to \mathsf{A}$$

$$\{$$
$$\mathsf{bord}'' :: \mathsf{D1} = \mathsf{D}$$
$$\mathsf{A} \lhd \mathsf{bord}' = \mathsf{D1}$$
$$\mathsf{A} \lhd \mathsf{next}' = \mathsf{add} \rhd (\mathsf{A} \lhd \mathsf{right}' \ /;/ \ \mathsf{A} \lhd \mathsf{up}')$$
$$\}$$

$$\begin{array}{lcl}
\mathsf{bord}'(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{t} \le \mathsf{n} \wedge (\mathsf{p} = \mathsf{n}{-}\mathsf{t}{+}1 \vee \mathsf{p} = \mathsf{t}{+}\mathsf{n}{-}1)) \ .(\mathsf{t},\mathsf{p}) \\
\mathsf{next}'(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{t} \le 2\mathsf{n}{-}1 \wedge \max(1, \mathsf{n}{-}\mathsf{t}{+}2, \mathsf{t}{-}\mathsf{n}{+}1) \le \mathsf{p} \le \min(2\mathsf{n}{-}1, \mathsf{t}{+}\mathsf{n}{-}2, 3\mathsf{n}{-}\mathsf{t}{-}1)) \\
 & & .(\mathsf{t},\mathsf{p}) \\
\mathsf{add}(\mathsf{a};\mathsf{b}) & = & (\mathsf{a}{+}\mathsf{b}) \\
\mathsf{right}'(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{t} \le 2\mathsf{n}{-}1 \wedge \max(1, \mathsf{n}{-}\mathsf{t}{+}2, \mathsf{t}{-}\mathsf{n}{+}1) \le \mathsf{p} \le \min(2\mathsf{n}{-}1, \mathsf{t}{+}\mathsf{n}{-}2, 3\mathsf{n}{-}\mathsf{t}{-}1) \\
 & & \wedge (\mathsf{p}{+}\mathsf{t}{-}\mathsf{n}{+}1)\%2 = 0)) \ .(\mathsf{t}{-}1, \mathsf{p}{-}1) \\
\mathsf{up}'(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{t} \le 2\mathsf{n}{-}1 \wedge \max(1, \mathsf{n}{-}\mathsf{t}{+}2, \mathsf{t}{-}\mathsf{n}{+}1) \le \mathsf{p} \le \min(2\mathsf{n}{-}1, \mathsf{t}{+}\mathsf{n}{-}2, 3\mathsf{n}{-}\mathsf{t}{-}1) \\
 & & \wedge (\mathsf{p}{+}\mathsf{t}{-}\mathsf{n}{+}1)\%2 = 0)) \ .(\mathsf{t}{-}1, \mathsf{p}{+}1) \\
\mathsf{bord}''(\mathsf{t},\mathsf{p}) & = & (1 \le \mathsf{t} \le \mathsf{n} \wedge (\mathsf{p} = \mathsf{n}{-}\mathsf{t}{+}1 \vee \mathsf{p} = \mathsf{t}{+}\mathsf{n}{-}1)) \ .(\mathsf{p}{+}\mathsf{t}{-}\mathsf{n}{+}1)/2, (\mathsf{t}{-}\mathsf{p}{+}\mathsf{n}{+}1)/2)
\end{array}$$

A ce stade, nous considérons que le processus de raffinement est terminé. Ce programme explicite un cadencement et une allocation des calculs : le calcul s'effectue en $2n{-}1$ étapes sur un réseau systolique linéaire à $2n{-}1$ processeurs dont les liens sont définis par les déplacements right' et up'.
◇

En parallélisation automatique, le raisonnement est basé sur la géométrie et requiert certaines hypothèses sur les domaines considérés. Les formalismes ALPHA et CRYSTAL, qui font la synthèse des transformations en parallélisation automatique, répercutent ces hypothèses

dans leur définition : en ALPHA, les variables sont dites *spatiales* et notent une collection de valeurs indicées dans un domaine polyèdrique convexe de $\mathbb{Z}^n$. Les énoncés sont des systèmes d'équations récurrentes affines. De tels énoncés peuvent être modélisés par un polytope et la construction de programmes parallèles peut être automatisée au sein de ce modèle, appelé *polytope model* [47]. En CRYSTAL, les variables sont appelées *champs de données* et notent des fonctions sur des domaines qui sont des graphes munis d'une *métrique de référence*.

Le cadre formel PEI permet de relâcher ces hypothèses en fournissant une abstraction de ces domaines sous la forme d'un objet PEI. Les objets PEI généralisent les variables en ALPHA ou CRYSTAL dans la mesure où la géométrie des objets PEI est seulement déterminée par une bijection (implicite) vers un sous-ensemble de $\mathbb{Z}^n$. Cette notion de géométrie est suffisante pour exprimer et appliquer avec sûreté une transformation qui consiste à changer de domaine de référence pour une variable.

L'abstraction fournie par PEI permet de dépasser les contraintes en parallélisation automatique. En particulier, dans [70], nous montrons que PEI permet de traiter un problème difficile en parallélisation automatique, le produit de Dirichlet qui présente des dépendances non-affines. Les techniques de transformations qui reposent sur la manipulation de polyèdres convexes échouent sur un tel exemple. Dans [14], les auteurs ont souligné ce point et ont développé un métalangage pour transformer des énoncés en CRYSTAL. Le traitement de cet exemple en PEI montre que le formalisme PEI contient des notions suffisantes pour mener à bien ces transformations.

1.3.2 Construction de programmes data-parallèles

Les programmes data-parallèles permettent d'exploiter certains opérateurs de communication disponibles sur certaines architectures SIMD comme par exemple la diffusion. Nous avons montré que le calcul de raffinement en PEI permet d'introduire de tels opérateurs en remplaçant une définition récursive correspondante à des communications point-à-point, par une définition non-récursive correspondante à une diffusion. En PEI, l'introduction d'une diffusion repose sur la propriété suivante dont la preuve peut être trouvée dans [34] :

Propriété 2 (Diffusion et récursivité) Soit une fonction dont une définition récursive est notée $\mathsf{g} = \mathsf{i} \,\#\, \mathsf{g} \circ \mathsf{s}$, où i note l'identité partiellement définie et s note une quelconque fonction. L'équation $\mathsf{Y} = \mathsf{X} \lhd \mathsf{g}$ est raffinée par l'équation : $\mathsf{Y} = \mathsf{X} \lhd \mathsf{i} \;/;/\; \mathsf{Y} \lhd \mathsf{s}$.

Exemple 3 (Produit matrice-vecteur)

Considérons l'énoncé suivant pour le calcul des produits élémentaires dans le produit matrice-vecteur :

$$
\begin{array}{l}
\mathsf{MatVec1}[\mathsf{n}] : \mathsf{A}, \mathsf{V} \rightarrow \mathsf{P} \\
\{ \\
\mathsf{A} = \mathsf{matrix} :: \mathsf{A} \\
\mathsf{diagonale} :: \mathsf{B0} = \mathsf{V} \\
\mathsf{B} = \mathsf{B0} \ /;/ \ (\mathsf{B} \lhd \mathsf{rowwise}) \\
\mathsf{P} = \mathsf{prod} \rhd (\mathsf{A} \ /;/ \ \mathsf{B}) \\
\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{matrix}(\mathsf{i},\mathsf{j}) & = & (0 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}{-}1) \,.(\mathsf{i},\mathsf{j}) \\
\mathsf{diagonale}(\mathsf{i},\mathsf{j}) & = & (0 \leq \mathsf{j} \leq \mathsf{n}{-}1 \wedge \mathsf{i} = \mathsf{j}) \,.(\mathsf{j}) \\
\mathsf{rowwise}(\mathsf{i},\mathsf{j}) & = & (0 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}{-}1 \wedge \mathsf{i} \neq \mathsf{j}) \,.((\mathsf{i}{-}1)\%\mathsf{n},\mathsf{j}) \\
\mathsf{prod}(\mathsf{a};\mathsf{b}) & = & (\mathsf{a}{\times}\mathsf{b})
\end{array}
$$

Les deux premières équations de cet énoncé expriment que le vecteur est placé sur la diagonale de la matrice. La troisième équation exprime des communications point-à-point : les valeurs du vecteur sont décalées à partir de la diagonale d'une position étape par étape selon la direction des colonnes (et en considérant cette dimension torique). Ces communications point-à-point sont exprimées par le déplacement défini par rowwise. Enfin, la quatrième équation exprime le calcul des produits élémentaires.

Dans l'énoncé ProdMat2 suivant, le déplacement défini par spread exprime une diffusion du vecteur à partir de la diagonale.

$$
\begin{array}{l}
\mathsf{MatVec2}[\mathsf{n}] : \mathsf{A}, \mathsf{V} \rightarrow \mathsf{P} \\
\{ \\
\mathsf{A} = \mathsf{matrix} :: \mathsf{A} \\
\mathsf{diagonale} :: \mathsf{B0} = \mathsf{V} \\
\mathsf{B} = \mathsf{B0} \lhd \mathsf{spread} \\
\mathsf{P} = \mathsf{prod} \rhd (\mathsf{A} \ /;/ \ \mathsf{B}) \\
\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{matrix}(\mathsf{i},\mathsf{j}) & = & (0 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}{-}1) \,.(\mathsf{i},\mathsf{j}) \\
\mathsf{diagonale}(\mathsf{i},\mathsf{j}) & = & (0 \leq \mathsf{j} \leq \mathsf{n}{-}1 \wedge \mathsf{i} = \mathsf{j}) \,.(\mathsf{j}) \\
\mathsf{spread} & = & \lambda(\mathsf{i},\mathsf{j}) \mid (0 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}{-}1) \,.(\mathsf{j},\mathsf{j}) \\
\mathsf{prod}(\mathsf{a};\mathsf{b}) & = & (\mathsf{a}{\times}\mathsf{b})
\end{array}
$$

Le raisonnement en PEI permet de montrer que l'énoncé MatVec2 est raffiné par MatVec1 (ce qui traduit un raffinement opérationnel : l'énoncé MatVec1 est en effet plus prescriptif dans la mesure où il décrit une implantation particulière de la diffusion).

D'après la propriété 2, la preuve consiste à montrer que spread peut se réécrire de la manière

suivante :

$$\begin{array}{rcl}
\mathsf{spread} & = & \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1)\ .(\mathsf{j},\mathsf{j}) \\
& = & \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1 \wedge \mathsf{i} = \mathsf{j})\ .(\mathsf{j},\mathsf{j}) \ \#\ \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1 \wedge \mathsf{i} \neq \mathsf{j})\ .(\mathsf{j},\mathsf{j}) \\
& = & \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1 \wedge \mathsf{i} = \mathsf{j})\ .(\mathsf{i},\mathsf{j}) \ \#\ \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1)\ .(\mathsf{j},\mathsf{j}) \\
& \circ & \lambda(\mathsf{i},\mathsf{j}) \mid (0 \le \mathsf{i},\mathsf{j} \le \mathsf{n}{-}1 \wedge \mathsf{i} \neq \mathsf{j})\ .((\mathsf{i}{-}1)\%\mathsf{n},\mathsf{j}) \\
& = & \mathsf{i}\ \#\ \mathsf{spread} \circ \mathsf{rowwise}
\end{array}$$

◇

Les programmes data-parallèles permettent aussi de préciser la localité des données par alignement des données sur une grille de processeurs virtuels. Dans [72], nous montrons comment le raffinement en Pei peut être utilisé pour modifier l'alignement des données afin de réduire le volume de communications. Les transformations utilisées reposent sur l'équivalence faible.

Nous avons illustré ce point sur le problème du calcul de l'opération *Gaxpy* [37] : $Ax + y$ avec A, une matrice et x et y, deux vecteurs, dans le cas particulier où $y = x$. Cette opération est souvent itérée pour calculer $A(...(A(Ax + x) + x)...) + x$, par exemple dans la méthode du gradient conjugué. Le formalisme Pei permet d'exprimer deux solutions data-parallèles pour ce problème. Les solutions diffèrent dans le placement du vecteur x : dans l'une des versions, le vecteur est placé selon un axe canonique de la matrice, dans l'autre, le vecteur est placé sur la diagonale. Ces deux solutions engendrent des diffusions et un volume de communications différents. Le raisonnement en Pei permet de montrer que l'une peut être déduite de l'autre.

1.3.3 Calcul creux

Le calcul creux est opéré notamment en algèbre linéaire lorsque l'on traite des matrices creuses. Une matrice est dite creuse lorsqu'elle contient une assez grande quantité de zéros. De telles matrices peuvent apparaître lors de la résolution d'équations à dérivées partielles en utilisant les méthodes à éléments finis par exemple. Elles requièrent un traitement particulier, étant donné que seuls les éléments non nuls ont besoin d'être stockés et que l'on peut économiser à la fois de l'espace mémoire et du temps de calcul [10].

Dans [74] et dans la thèse de Frédérique Voisin [73], nous montrons que Pei permet de traiter le calcul creux en adaptant un programme de façon à respecter un stockage optimal des structures de données creuses. Un tel stockage diffère d'un accès mémoire naturel puisque l'emplacement des éléments non nuls de la structure doit être déterminé. En Pei, cela signifie que le stockage optimal et la structure creuse sans les zéros sont deux objets Pei fortement équivalents. Le réindiçage d'un objet Pei définit la façon dont les éléments non nuls de la structure creuse sont stockés. Transformer le code consiste donc à appliquer un changement de base au programme dense.

1.3.4 Autres stratégies de construction de programmes

Le changement de base est un exemple de schéma type de transformations constitué de l'application successive de plusieurs règles élémentaires. De tels schémas définissent des stratégies permettant de transformer un énoncé sous une certaine forme en un énoncé sous une autre forme et démontrer leur équivalence. Sur la base des règles de raffinement en Pei, nous

avons développé de nouvelles stratégies de transformations. Quelques unes de ces stratégies sont présentées dans [34] et dans la thèse de Stéphane Genaud [33]. Ces stratégies visent à atteindre des objectifs particuliers comme par exemple la minimisation des mouvements de données. Parmi ces stratégies, citons par exemple, la stratégie de *simplification des communications* qui vise à simplifier les déplacements en trouvant un réindiçage adéquat : lorsque que le changement de base défini par ce réindiçage est appliqué, l'énoncé obtenu présente un schéma de communications plus simple. Cette stratégie s'applique à condition que chacune des fonctions g qui définissent un déplacement puisse s'écrire sous la forme générale $g(i_1, \ldots, i_m, j) = ((i_1 + a_1 j + b_1)\%n, \ldots, (i_m + a_m j + b_m)\%n, j-1)$, ce qui représente une classe importante d'algorithmes systoliques.

1.4 Conclusion

Les techniques de programmation parallèle fiables sont fondées sur des transformations formelles d'énoncés. Le cadre formel PEI a été conçu pour appliquer ces transformations avec sûreté, pour exprimer les énoncés successifs et les règles de transformation elles-mêmes.

La notion d'objet PEI, au centre du cadre formel, intégre une notion abstraite de géométrie sous la forme d'une bijection vers un placement de référence virtuel. Cette géométrie relie le domaine du problème avec le domaine de l'architecture.

Cette notion d'objet permet de ménager toutes les transformations qui consistent en des manipulations géométriques des objets du problème. On généralise ainsi l'idée en parallélisation automatique qui propose de substituer un domaine espace-temps au domaine de calcul initial.

Nos travaux ont montré que PEI est un bon cadre pour la conception de programmes parallèles.

Chapitre 2

Un domaine sémantique pour les langages data-parallèles

2.1 Introduction

A notre sens, tout paradigme de programmation parallèle ([57] en donne une classification) définit un certain degré de coopération entre le programmeur et le compilateur pour obtenir une implantation efficace : à une extrémité du spectre, le passage de messages PVM ou MPI qui laisse le programmeur en charge de tout le travail, à l'autre extrémité, la parallélisation automatique où le compilateur est le seul moteur du raisonnement. Entre ces deux extrémités, le paradigme émergeant associé au standard OpenMP [56] et le data-parallélisme.

Bien que OpenMP soit plus récent que les langages data-parallèles standards C* ou Hpf, à notre avis, il définit un degré de coopération qui n'est pas optimal, principalement parce qu'il ne propose pas de facilité pour exprimer la localité des données. Ceci peut affecter les performances de manière critique sur des machines qui exhibent des temps d'accès mémoire non uniformes. Les recherches récentes qui tendent à étendre OpenMP avec des idées héritées du data-parallélisme [45, 13] soulignent l'importance des expressions de la localité des données. OpenMP est plus dépendant de la machine. C'est pourquoi le programmeur OpenMP a plus d'effort à fournir pour apporter ces connaissances au compilateur OpenMP afin d'atteindre l'efficacité des calculs. Ceci est confirmé par les travaux récents qui ont pour but de générer du code OpenMP à partir de programmes Hpf [8] ou de compiler des programmes Hpf pour des environnements d'exécution multi-threadés [4]. En quelque sorte, ces travaux montrent que le data-parallélisme pourrait définir une meilleure coopération entre le programmeur et le compilateur.

Le data-parallélisme possède d'indéniables qualités en terme de facilité pour le programmeur à concevoir un programme paralléle. Un intérêt majeur du data-parallélisme est qu'il autorise le programmeur à décrire un algorithme paralléle en choisissant un algorithme séquentiel bien connu et en concentrant ses efforts sur l'expression de la localité des données. En un sens, cela révèle que les concepts pour exprimer la localité des données sont de première importance en programmation paralléle et peuvent même être considérés comme l'«essence du parallélisme» en comparaison de la programmation séquentielle.

Cependant, restent des questions cruciales pour la définition et la maîtrise du data-parallé-

lisme : parmi ces questions comment transformer avec sûreté un programme écrit dans l'un des deux langages data-parallèles standards HPF [44] et C* [60] : il semble assez difficile de conduire un raisonnement menant à une implantation efficace avec de tels langages impératifs. De plus, HPF et C* utilisent des sémantiques intuitives différentes pour l'expression de la localité des données : les concepts d'*alignement* en HPF et de *shape* en C*. Nous sommes donc en droit de nous poser la question suivante : qu'est-ce que le data-parallélisme ? [67] HPF ou C* ? Il est intéressant de noter que ces deux concepts introduisent un déséquilibre dans le partage du travail entre le programmeur et le compilateur pour obtenir une implantation efficace : en C*, le programmeur a beaucoup d'efforts à faire pour adapter un programme séquentiel et le travail du compilateur est plus facile. En HPF, le programmeur a relativement peu de travail en partant d'un programme séquentiel, mais beaucoup de travail reste à la charge du compilateur.

Ces différences et ce déséquilibre justifient l'introduction d'un domaine sémantique pour les langages data-parallèles où les deux concepts de localité des données en HPF et C* pourraient être expliqués et situés l'un relativement à l'autre. Un tel domaine permettrait de répondre aux interrogations au sujet du data-parallélisme et établirait une meilleure coopération entre le programmeur et le compilateur. Nos recherches se sont portées sur la définition de ce domaine sémantique sur la base de nos travaux sur les transformations d'énoncés.

Le formalisme PEI et les langages data-parallèles ont beaucoup de similitudes bien que d'origines différentes : PEI est clairement issu d'une généralisation des techniques en parallélisation automatique, tandis que les premiers langages data-parallèles ont été conçus pour programmer les machines SIMD. Nous avons étudié ces similitudes et proposé un domaine sémantique issu de PEI où les concepts d'alignement et de shape peuvent se rejoindre.

Nous avons montré que ce domaine sémantique est bien adapté pour donner une sémantique aux expressions de la localité des données dans les langages data-parallèles et nous avons développé quelques exemples visant à montrer que ce domaine sémantique permet d'établir une meilleure coopération entre le programmeur et le compilateur que les langages data-parallèles standards.

Nous avons également étudié la sémantique des directives telles qu'elles sont définies dans le langage HPF. Ces directives sont de simples conseils au compilateur qui concernent seulement l'efficacité des calculs sans remettre en cause la correction des programmes. Nous avons considéré l'utilisation de cette notion de directive comme un outil de programmation pour faciliter la coopération entre le programmeur et le compilateur.

Ce chapitre présente ces travaux concernant la définition du data-parallélisme.

2.2 PEI et les langages data-parallèles

Dans [53, 52], nous mettons en évidence que toutes les notions de PEI ont un lien avec les primitives caractéristiques des langages data-parallèles : l'alignement de données sur une grille de processeurs virtuels ressemble à une composition du réindiçage et de la superposition, les opérations globales sont définies par l'application d'un calcul à un objet PEI, la diffusion et la communication par décalage ou globale sont des déplacements particuliers, la réduction est une caractéristique intrinsèque de PEI. Ces différents points sont précisés dans la suite.

Nous avons cherché à éclairer ces similitudes en étudiant le passage d'un énoncé PEI à un

programme data-parallèle. Un prototype de traducteur a été réalisé pour générer du code HPF à partir de PEI [33].

2.2.1 Alignement de données

Les expressions de la localité des données sont essentielles dans les langages data-parallèles car ce sont elles qui déterminent l'usage de la mémoire et les communications sur l'architecture physique.

Exemple 4 Extrait du produit de deux matrices carrées.

$$
\begin{array}{l}
\mathsf{ProdMat} : \mathsf{A}, \mathsf{B} \rightarrow \mathsf{C} \\
\{ \\
\mathsf{align\_j} :: \mathsf{A}' \;=\; \mathsf{A} \\
\mathsf{align\_i} :: \mathsf{B}' \;=\; \mathsf{B} \\
\mathsf{P} \qquad\quad =\; \ldots\ ((\mathsf{A}' \lhd \ldots)\ /;/\ (\mathsf{B}' \lhd \ldots))\ \ldots \\
\ldots \\
\} \\
\mathsf{align\_j} \;=\; \lambda(\mathsf{i},\mathsf{j},\mathsf{k}) \mid (1 \leq \mathsf{i},\mathsf{k} \leq \mathsf{n} \wedge \mathsf{j} = 1)\ .(\mathsf{i},\mathsf{k}) \\
\mathsf{align\_i} \;=\; \lambda(\mathsf{i},\mathsf{j},\mathsf{k}) \mid (1 \leq \mathsf{j},\mathsf{k} \leq \mathsf{n} \wedge \mathsf{i} = 1)\ .(\mathsf{k},\mathsf{j})
\end{array}
$$

Ces équations et expressions décrivent le même placement de données que le programme HPF ci-dessous qui utilise des *directives d'alignement* et un *template* [28]. Ce programme indique que les deux matrices sont alignées sur deux faces d'un cube :

```
!HPF$     TEMPLATE CUBE(N,N,N)
          DIMENSION (N,N) :: A,B

!HPF$     ALIGN A(I,K) WITH CUBE(I,1,K)
!HPF$     ALIGN B(K,J) WITH CUBE(1,J,K)
          ...
```

◇

Si deux objets PEI ont leurs valeurs associées à des coordonnées dans un même espace géométrique ($\mathbb{Z}^n$), alors leur superposition exprime des *collections* de valeurs de même *shape* selon la terminologie C*.

Exemple 5 Somme de deux matrices carrées.

$$
\begin{array}{l}
\mathsf{SumMat} : \mathsf{A}, \mathsf{B} \to \mathsf{C} \\
\{ \\
\mathsf{A} \;=\; \mathsf{matrix} :: \mathsf{A} \\
\mathsf{B} \;=\; \mathsf{matrix} :: \mathsf{B} \\
\mathsf{C} \;=\; \mathsf{add} \triangleright (\mathsf{A} \;/;/\; \mathsf{B}) \\
\} \\
\mathsf{matrix} \;=\; \lambda(\mathsf{i},\mathsf{j}) \mid (1 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n})\ .(\mathsf{i},\mathsf{j}) \\
\mathsf{add} \;\;\;=\; \lambda(\mathsf{a};\mathsf{b})\,.\,(\mathsf{a}+\mathsf{b})
\end{array}
$$

L'objet (A /;/ B) représente une grille de processeurs virtuels qui stocke les valeurs locales de A et B. Le résultat C est aligné sur la même *shape* et n^2 additions peuvent s'éxécuter en parallèle, comme le signifie ce code C* [60] :

```
shape [n][n]matrix;
float:matrix A,B,C;

C=A+B;
```

◇

2.2.2 Opérations globales

L'opération nommée «calcul» en Pei est clairement une opération globale sur tous les éléments d'un objet Pei : il applique une fonction partielle f, notée en Pei $\lambda \mathsf{x} \mid (\mathsf{P}(\mathsf{x}))\ .\mathsf{f}(\mathsf{x})$, sur tous les éléments dont le type est scalaire ou une séquence de scalaires s'ils résultent d'une superposition. La domaine de f, défini par le prédicat P(x), exprime une *sélection* de la même façon qu'une instruction `where` dans un langage data-parallèle.

Exemple 6

$$
\begin{array}{l}
\mathsf{Sup} : \mathsf{A}, \mathsf{B} \to \mathsf{C} \\
\{ \\
\mathsf{A} \;=\; \mathsf{vector} :: \mathsf{A} \\
\mathsf{B} \;=\; \mathsf{vector} :: \mathsf{B} \\
\mathsf{C} \;=\; \mathsf{test} \triangleright (\mathsf{A} \;/;/\; \mathsf{B}) \\
\} \\
\mathsf{vector} \;=\; \lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}) \\
\mathsf{test} \;\;\;=\; \lambda(\mathsf{a};\mathsf{b}) \mid (\mathsf{a} \neq 0 \wedge \mathsf{b} \neq 0)\ .\mathsf{max}(\mathsf{a},\mathsf{b})
\end{array}
$$

En supposant que `A` et `B` sont alignés sur un tableau monodimensionnel, l'énoncé ci-dessus est exprimé dans un langage data-parallèle par :

```
where((A!=0) && (B!=0))
      C = max(A,B);
```

◇

2.2.3 Communications et diffusions

Des exemples précédents ont mis en évidence des déplacements en PEI : certains de ces exemples définissaient des dépendances régulières, telles que les translations uniformes. D'autres présentaient des diffusions à travers des fonctions non injectives. De telles déplacements apparaissent aussi dans l'exemple 1 :

$$\mathsf{P} = \ldots\ ((\mathsf{A}' \lhd \mathsf{spread\_i})\ /;/\ (\mathsf{B}' \lhd \mathsf{spread\_j}))\ \ldots$$

avec :

$$\begin{array}{rcl} \mathsf{spread\_j} & = & \lambda(\mathsf{i},\mathsf{j},\mathsf{k}) \mid (1 \le \mathsf{i},\mathsf{j},\mathsf{k} \le \mathsf{n})\ .(\mathsf{i},1,\mathsf{k}) \\ \mathsf{spread\_i} & = & \lambda(\mathsf{i},\mathsf{j},\mathsf{k}) \mid (1 \le \mathsf{i},\mathsf{j},\mathsf{k} \le \mathsf{n})\ .(1,\mathsf{j},\mathsf{k}) \end{array}$$

Un tel déplacement non injectif est exprimé par des primitives de diffusion dans les langages data-parallèles. Cet exemple utiliserait des notations comme [61] :

```
SPREAD(A,DIM=2,NCOPIES=N)
SPREAD(B,DIM=1,NCOPIES=N)
```

Inversement, les translations uniformes en PEI sont exprimées par des communications implicites dans les langages data-parallèles. Par exemple, l'équation :

$$\mathsf{R} = \mathsf{D} \lhd \lambda(\mathsf{i},\mathsf{j},\mathsf{k}) \mid (1 \le \mathsf{i},\mathsf{j} \le \mathsf{n} \wedge 1 \le \mathsf{k} < \mathsf{n})\ .(\mathsf{i},\mathsf{j},\mathsf{k}{+}1)$$

peut s'exprimer par :

```
R = [.][.][.+1]D;
```

2.2.4 Scan et réduction

La notion de *réduction* est un autre concept majeur dans les langages data-parallèles. Considérons le problème de la somme de n nombres $s = \sum_{i \in 1..n} a_i$. Une solution s'obtient en utilisant un déplacement pour décaler les valeurs d'un objet monodimensionnel d'une position, étape par étape.

$$\mathsf{Sum} : \mathsf{A} \to \mathsf{S}$$

$$\begin{array}{l} \{ \\ \mathsf{A} = \mathsf{vector} :: \mathsf{A} \\ \mathsf{T} = \mathsf{add} \rhd (\mathsf{A}\ /;/\ \mathsf{T} \lhd \mathsf{pre}) \\ \mathsf{S} = \mathsf{T} \lhd \mathsf{last} \\ \} \end{array}$$

$$\begin{array}{rcl} \mathsf{vector} & = & \lambda(\mathsf{i}) \mid (1 \le \mathsf{i} \le \mathsf{n})\ .(\mathsf{i}) \\ \mathsf{pre} & = & \lambda(\mathsf{i}) \mid (1 < \mathsf{i} \le \mathsf{n})\ .(\mathsf{i}-1) \\ \mathsf{last} & = & \lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i}) \\ \mathsf{add} & = & \lambda(\mathsf{a};\mathsf{b})\,.\,\mathsf{a}{+}\mathsf{b} \end{array}$$

Comme nous le verrons dans la suite, ceci est une façon particulière d'implanter la somme de n nombres qui est une autre caractéristique classique des langages data-parallèles : le calcul préfixe par une fonction `scan`. Cette définition s'obtient par raffinement d'une définition non-déterministe plus générale.

De manière générale, la réduction signifie l'implantation d'une opération n-aire en utilisant une opération binaire. Au niveau d'une spécification ou d'une programmation haut-niveau, l'utilisateur ne devrait pas préciser le choix d'implantation. Sur notre exemple, puisque l'addition est associative et commutative, la seule chose à indiquer est que les n valeurs doivent être regroupées, peu importe l'ordre de la séquence pour former un objet à une seule valeur. La fonction définissant cette sorte de regroupement n'est pas injective, mais elle doit être définie au moyen de fonctions injectives de façon a être implantée : en PEI, une telle implantation s'exprime en utilisant des déplacements.

Ainsi en PEI, une réduction est l'inverse d'un déplacement. Nous choisissons de noter son application par $\mathsf{g} \,;\triangleright\, \mathsf{X}$ et elle forme en tout point de coordonnées z, une séquence (dans un ordre quelconque) des valeurs de l'objet noté X associées aux points de coordonnées y telles que z est l'image de y par la fonction notée g.

Définition 11 (Réduction) Soit $X = (v : \sigma)$, un objet PEI. Soit g, une fonction partielle : $\mathbb{Z}^n \rightharpoonup \mathbb{Z}^n$ telle que $\mathrm{img}(g) \subseteq \mathrm{def}(\sigma)$ et $\mathrm{def}(g) \subseteq \mathrm{def}(v)$. Une réduction de X par g est un objet PEI $(w : \sigma)$ tel que :

- $\mathrm{def}(w) = g(\mathrm{def}(v))$
- $w(z)$ est n'importe quelle séquence formée des valeurs de $\prec v(y) \mid g(y) = z \succ$.

Propriété 3 (Réduction et déplacement)

$$\mathsf{g} \,;\triangleright\, \mathsf{X} \;=\; \mathsf{X} \triangleleft \mathsf{inv(g)} \text{ ssi } \mathsf{g} \text{ note une fonction injective.}$$

Les autres propriétés de la réduction se traduisent par les règles de raffinement suivantes :

$$\mathsf{X} \,;\triangleright\, (\mathsf{g1}\#\mathsf{g2}) \;\sqsubseteq\; (\mathsf{X} \,;\triangleright\, \mathsf{g1}) \;/;/\; (\mathsf{X} \,;\triangleright\, \mathsf{g2})$$
$$\mathsf{X} \,;\triangleright\, (\mathsf{g1} \circ \mathsf{g2}) \;\sqsubseteq\; (\mathsf{X} \,;\triangleright\, \mathsf{g1}) \,;\triangleright\, \mathsf{g2}$$

où $\mathsf{g1}\#\mathsf{g2}$ note une fonction définie sur deux sous-domaines disjoints.

Revenons au problème de la somme de n nombres. Ce problème est exprimé par l'énoncé suivant :

$$\mathsf{Sum} : \mathsf{A} \to \mathsf{S}$$
$$\{$$
$$\mathsf{A} \;=\; \mathsf{vector} :: \mathsf{A}$$
$$\mathsf{S} \;=\; \mathsf{rec\_add} \triangleright (\mathsf{gather} \,;\triangleright\, \mathsf{A})$$
$$\}$$
$$\mathsf{vector} \;=\; \lambda(\mathsf{i}) \mid (1 \le \mathsf{i} \le \mathsf{n}) \,.(\mathsf{i})$$
$$\mathsf{gather} \;=\; \lambda(\mathsf{i}) \mid (1 \le \mathsf{i} \le \mathsf{n}) \,.(\mathsf{n})$$

La réduction dans la seconde équation associe au point de coordonnée n, une séquence de toutes les valeurs de A. Ces valeurs sont ensuite additionnées par la fonction rec_add dont la définition récursive n'est pas donnée ici.

Cet énoncé avec une réduction de A par gather peut être raffiné en introduisant une définition récursive de la fonction gather :

$$\mathsf{gather} = \lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i})\ \#\ (\mathsf{gather} \circ \lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{+}1))$$

Sachant que $\lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{-}1)$ note une fonction partielle injective dont l'inverse peut être notée $\lambda(\mathsf{i}) \mid (1 < \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{+}1)$, nous pouvons écrire :

$$\begin{array}{ll}
& \mathsf{gather}\ ;\triangleright\ \mathsf{A} \\
\sqsubseteq & (\lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i})\ ;\triangleright\ \mathsf{A})\ \ /;/\ \ ((\mathsf{gather} \circ \lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{+}1))\ ;\triangleright\ \mathsf{A}) \\
\sqsubseteq & (\lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .\mathsf{i}\ ;\triangleright\ \mathsf{A})\ \ /;/\ \ (\mathsf{gather}\ ;\triangleright\ (\lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{+}1)\ ;\triangleright\ \mathsf{A})) \\
= & (\mathsf{A} \triangleleft \lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i}))\ \ /;/\ \ (\mathsf{gather}\ ;\triangleright\ (\mathsf{A} \triangleleft \lambda(\mathsf{i}) \mid (1 < \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{-}1)))
\end{array}$$

En posant $\mathsf{pre} = \lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}{-}1)$ et $\mathsf{last} = \lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i})$, la dernière expression est :

$$(\mathsf{A} \triangleleft \mathsf{last})\ \ /;/\ \ (\mathsf{gather}\ ;\triangleright\ (\mathsf{A} \triangleleft \mathsf{pre}))$$

La même raisonnement mène à l'expression suivante :

$$(\mathsf{A} \triangleleft \mathsf{last})\ \ /;/\ \ ((\mathsf{A} \triangleleft \mathsf{pre}) \triangleleft \mathsf{last})\ \ /;/\ \ (\mathsf{gather}\ ;\triangleright\ (\mathsf{A} \triangleleft \mathsf{pre}))$$

etc. Nous reconnaissons alors une définition récursive de l'objet A, en utilisant cette propriété :

Propriété 4 (Réduction et récursivité) Soit une fonction dont la définition récursive est notée $\mathsf{f} = \mathsf{i}\ \#\ \mathsf{f} \circ \mathsf{inv}(\mathsf{s})$, où i note l'identité et s note une fonction partielle injective. L'équation $\mathsf{Y} = \mathsf{f}\ ;\triangleright\ \mathsf{X}$ est raffinée par les deux équations suivantes, où T est une variable intermédiaire définie récursivement :

$$\begin{array}{l}
\mathsf{T} = \mathsf{X}\ \ /;/\ \ (\mathsf{T} \triangleleft \mathsf{s}) \\
\mathsf{Y} = \mathsf{T} \triangleleft \mathsf{i}
\end{array}$$

.

Cette propriété mène à l'énoncé suivant pour la somme de n nombres :

$$\begin{array}{l}
\mathsf{Sum} : \mathsf{A} \rightarrow \mathsf{S} \\
\{ \\
\mathsf{A} \ = \ \mathsf{vector} :: \mathsf{A} \\
\mathsf{T} \ = \ \mathsf{A}\ \ /;/\ \ \mathsf{T} \triangleleft \mathsf{pre} \\
\mathsf{S} \ = \ \mathsf{rec_add} \triangleright (\mathsf{T} \triangleleft \mathsf{last}) \\
\} \\
\\
\mathsf{vector} \ = \ \lambda(\mathsf{i}) \mid (1 \leq \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i}) \\
\mathsf{pre} \ = \ \lambda(\mathsf{i}) \mid (1 < \mathsf{i} \leq \mathsf{n})\ .(\mathsf{i} - 1) \\
\mathsf{last} \ = \ \lambda(\mathsf{i}) \mid (\mathsf{i} = \mathsf{n})\ .(\mathsf{i})
\end{array}$$

Enfin, une addition scalaire peut être substituée à la fonction récursive rec_add définie sur la séquence de valeur au point de coordonnée n. Cela mène au premier programme de cette section.

Cela termine le processus de raffinement pour transformer le programme précédent en introduisant une réduction. Un tel procédé pourrait être utilisé par un compilateur pour générer du code pour de telles opérations macroscopiques. Bien sûr, d'autres raffinements peuvent être proposés, qui peuvent exprimer d'autres implantations parallèles efficaces d'une réduction, en définissant d'autres domaines de référence et d'autres définitions récursives.

2.3 Différentes sémantiques intuitives

Nous avons vu que le cadre formel PEI permet d'exprimer de manière abstraite la plupart des caractéristiques des langages data-parallèles. Un grand intérêt de PEI vis-à-vis de ces langages est qu'il autorise un raisonnement sur les énoncés basé sur des transformations exprimées dans le même formalisme, et permettant d'atteindre avec sûreté des implantations efficaces.

Par ailleurs, les langages data-parallèles standards HPF et C* ont différentes sémantiques intuitives : ils utilisent différents concepts pour exprimer la localité des données. Ces concepts sont l'alignement en HPF et la *shape* (ou forme) en C*. Nous avons observé ces différences et montré qu'elles réduisaient la capacité de partage de travail entre le compilateur et le programmeur. En voici l'illustration à travers quelques exemples en HPF et C*.

Exemple 7 (Somme de vecteurs non-alignés)

```
      REAL A(0:7),B(0:7),C(0:7)
!HPF$ TEMPLATE X(0:14)
!HPF$ ALIGN A(I) WITH X(I+1)
!HPF$ ALIGN B(I) WITH X(2*I)
      ...
      C = A + B
      ...
```

où le template X est utilisé pour définir l'alignement de la variable A relativement à la variable B. L'alignement de la variable C est laissé à la charge du compilateur.

Le précédent programme HPF pourrait être codé comme suit en C* :

```
shape [15]vector;
real: vector A,B,C;
...
where (pc_coord(0)<8)
  C = [.+1]A + [.*2]B;
...
```

où A, B sont «ré-indicés» dans l'affectation de façon à décrire une relation entre valeurs de vecteur et processeurs virtuels équivalente à celle décrite par les directives d'alignement dans le programme HPF.

En C*, les communications (virtuelles) entre processeurs virtuels sont exprimées explicitement par le biais de l'affectation. De plus l'«alignement» de `C` ainsi que les indices des valeurs qui sont affectées sont définis par le programmeur qui doit aussi préciser quel sont les *processeurs virtuels actifs*.

La traduction de cet exemple simple de HPF vers C* révèle quelques unes des difficultés à surmonter à la fois par le compilateur de HPF pour obtenir un code efficace et par le programmeur en C* pour écrire son programme.
◇

Exemple 8 (Produit de matrices)

Voici un autre programme HPF typique d'une certaine façon de programmer en HPF car directement inspiré d'un code séquentiel :

```
      REAL A(0:7,0:7),B(0:7,0:7),C(0:7,0:7)
!HPF$ ALIGN A(I,*) WITH C(I,*)
!HPF$ ALIGN B(*,J) WITH C(*,J)
      ...
      DO K=0,7
        FORALL (I=0:7,J=0:7)
          C(I,J) = A(I,K)*B(K,J) + C(I,J)
        END FORALL
      END DO
      ...
```

Les directives de ce programme conseillent le compilateur de placer tous les éléments de la ligne `I` de la matrice `A` et tous les éléments de la colonne `J` de la matrice `B` sur le même processeur virtuel que l'élément `C(I, J)`. Il s'agit d'un cas particulier d'alignement appelé *collapsing*.

Dans cet exemple, étant donné que l'alignement est défini pour tout élément de `C`, toute ligne `I` de `A` est répliquée sur tout processeur virtuel correspondant à un élément de la ligne `I` de `C`. De façon similaire toute colonne `J` de `B` est répliquée sur tout processeur virtuel correspondant à un élément de la colonne `J` de `C`. Donc ce placement implique une grande quantité de *répliquations* : si le compilateur suit ces directives aucune communication ne sera réalisée. Cependant cela entrainera un usage mémoire probablement excessif en raison de la réplication de toutes les lignes et colonnes, à moins qu'une directive `DISTRIBUTE` soit insérée qui peut réduire l'espace mémoire nécessaire sur l'architecture physique. L'usage de telles directives peut devenir vraiment délicat pour le programmeur puisqu'elles peuvent avoir de lourdes conséquences sur le comportement du programme selon la façon dont elles interagissent ou la façon dont le compilateur les implante.

Sans plus d'aide de la part du programmeur, il est très difficile pour le compilateur de trouver un bon compromis entre communications et usage mémoire. En effet cela exige d'analyser, non seulement les directives d'alignement, mais aussi l'ordre de calcul de la somme des produits en utilisant la commutativité et l'associativité de l'addition.

Le programmeur peut toutefois aider un peu plus le compilateur en insérant une directive `INDEPENDENT` juste avant la boucle dans le programme. Cela indiquera au compilateur que les

pas de la boucle peuvent être exécutés dans n'importe quel ordre. Même si cette directive est ajoutée, beaucoup de travail reste à faire pour le compilateur afin de produire un code où les données sont distribuées au mieux.

Considérons maintenant la façon dont le langage C* peut aider à construire un programme performant pour le produit de matrices et quel partage du travail entre programmeur et compilateur il définit.

Le *collapsing* n'existe pas en C* mais le programmeur est responsable de l'usage mémoire et les communications sont explicites : le choix d'une forme pour les matrices induit un comportement du programme. Par exemple, pour calculer les produits localement, les matrices peuvent être placées dans une forme tridimensionnelle : leurs éléments diffusés aux bonnes positions et les produits calculés en place sans d'autres communications. Un autre choix possible correspond à l'algorithme de Cannon [46], et mène au programme suivant où les matrices sont placées dans une forme bidimensionnelle :

```
shape [8,8]matrix;
real: matrix A, B, C
...
A = [.,(.+pc_coord(0))%%8]A;
B = [(.+pc_coord(1))%%8,.]B;
...
for(k=0;k<8;k++)
  {
   A = [.,(.-1)%%8]A;
   B = [(.-1)%%8,.]B;
   C += A*B;
  }
...
```

Le programme ci-dessus décrit une version du produit de matrices où la mémoire est utilisée efficacement. Rappelons brièvement cet algorithme : les éléments de `A` et `B` sont préalablement réarrangés de façon à ce que les éléments `A(I,(I+J)%8)` et `B((I+J)%8,J)` soient placés sur le même processeur virtuel. Cet arrangement est réalisé en décalant tous les éléments de `A` de `I` positions en suivant la dimension torique des colonnes et de façon similaire, en décalant tous les éléments de `B` de `J` positions en suivant la dimension torique des lignes. Puis, à chaque itération, les éléments de `A` (resp. `B`) sont déplacés d'une position selon la même direction mais dans le sens contraire de façon à ce que les produits soient effectués localement.

Beaucoup de travail doit être accompli par le programmeur pour construire ce programme et particulièrement pour prouver sa correction. D'autre part, l'équilibrage de charge a été réalisé de tel sorte que la compilateur peut facilement produire un code efficace.

Cet exemple met clairement en évidence différents niveaux de coopération entre le programmeur et le compilateur. Maintenant la question est : est-ce que le programmeur est capable de transformer le programme HPF précédent de façon à ce que le compilateur puisse découvrir l'algorithme de Cannon ? Cela supposerait que HPF autorise des alignements non-linéaires afin qu'il soit possible d'exprimer l'arrangement initial des éléments des matrices `A` et `B`. Le programme pourrait alors se réécrire ainsi :

```
REAL A(0:7,0:7),B(0:7,0:7),C(0:7,0:7)
```

```
!HPF$ ALIGN A(I,J) WITH C(I,MODULO(J-I,8))
!HPF$ ALIGN B(I,J) WITH C(MODULO(I-J,8),J)
      ...
      DO K=0,7
        FORALL (I=0:7,J=0:7)
          C(I,J) = A(I,MODULO(I+J-K,8))*B(MODULO(I+J-K,8),J) + C(I,J)
        END FORALL
      END DO
      ...
```

où les pseudo-directives `ALIGN` spécifient le bon arrangement initial des éléments de matrices. Au-delà des différences syntaxiques entre HPF et C⋆ et considérant l'utilisation d'alignements non-linéaires, en HPF elle supposerait de définir de nouvelles techniques de compilation [3] et par conséquent une difficulté accrue pour le compilateur, tandis qu'en C⋆ elle impliquerait plus d'investissement de la part du programmeur pour écrire son programme.
◇

2.4 Notre domaine sémantique

Très peu de travaux sont consacrés à la sémantique des langages data-parallèles : nous pouvons mentionner le langage $\mathcal{L}$ [11] et sa sémantique qui définit un modèle pour les langages tels que C⋆. Nous pouvons également citer le modèle des *structures concrètes distribuées* [39] et le BSλ-calcul [49] qui définissent des notions de localité plus concrètes au sens d'un placement explicite des données sur les processeurs.

Les différences sémantiques des standards data-parallèles HPF et C⋆ justifient l'introduction d'un domaine sémantique où ces différences pourraient être expliquées et se rejoindre. Nous pensons que ce domaine sémantique établira une meilleur coopération entre le programmeur et le compilateur.

Nous avons proposé un tel domaine sémantique pour les expressions de la localité de données. Nous l'avons défini en reprenant le cadre formel PEI qui permet d'exprimer une certaine notion de localité. Nous montrons ci-après que PEI ne suffit pas à définir ce domaine sémantique car la notion de localité qu'il définit n'est pas suffisamment abstraite pour expliquer celle de HPF.

2.4.1 Vers une notion de localité plus abstraite

Le cadre formel PEI ne permet pas d'exprimer un certain type d'alignement en HPF : l'alignement avec répliquation.

Exemple 9 Considérons ces directives d'alignements en HPF pour le produit de matrices :

```
!HPF$     TEMPLATE CUBE(N,N,N)
          DIMENSION (N,N) :: A,B

!HPF$     ALIGN A(I,K) WITH CUBE(I,*,K)
```

```
!HPF$     ALIGN B(K,J) WITH CUBE(*,J,K)
          ...
```

Les matrices `A` et `B` sont répliquées dans un cube de coté n, de sorte que les éléments `A(I,K)` et `B(K,J)` se trouvent alignés i.e. en même position dans le cube. Cet alignement est tel que le calcul des produits élémentaires `A(I,K)` $\times$ `B(K,J)` ne requiert aucune communication.

Le cadre formel Pei ne permet pas d'exprimer un tel alignement. En effet, cet alignement implique que l'élément `A(I,K)` est aligné avec plusieurs éléments de `B` : tous les élements `B(K,J)` pour `J` $\in$ `1..N`. Or, en Pei, il n'est pas possible d'aligner une valeur d'un objet, disons A, avec plusieurs valeurs d'un autre objet, disons B. Cela supposerait que plusieurs valeurs de B aient le même indice. Or, chaque valeur d'un objet Pei a un indice unique dans le domaine de référence de l'objet.

◇

Notre domaine sémantique définit une notion de localité plus abstraite que celle de Pei.

2.4.2 Des objets «mis en forme»

Afin de définir une notion de localité plus abstraite que celle de Pei, notre domaine sémantique inclue une notion d'objets, appelés *objets mis en forme* qui généralisent les objets Pei.

Considérant un objet Pei, chacune de ces valeurs est associée à un nuplet d'entiers. On peut distinguer deux connotations pour un tel nuplet : celle d'*indice* : sorte d'étiquette permettant de référencer une valeur de manière unique, et celle de *position* : coordonnées d'un point dans un espace géométrique discret muni d'un repère.

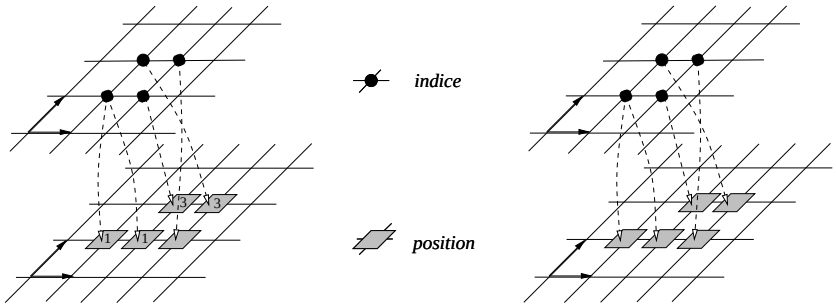
En un certain sens, en Pei, ces deux connotations peuvent être séparées : le nuplet z associé à la valeur par le placement arbitraire joue le rôle d'indice et le nuplet $\sigma(z)$ associé à la valeur par le placement de référence joue le rôle de position. Dans ce sens, la relation entre indices et positions d'un objet Pei est définie par une bijection – la bijection σ – et une valeur admet donc une seule position.

Les *objets mis en forme* généralisent les objets Pei au sens où la relation entre indices et positions d'un objet mis en forme est définie par une fonction quelconque de l'ensemble des positions vers l'ensemble des indices, de sorte qu'une valeur peut admettre plusieurs positions. Chaque valeur possède un ou plusieurs *représentant(s)* en chacune de ses positions (cf. figure 2.1(a)).

Un objet mis en forme peut être vu comme un conteneur de données (les représentants). Dans cette interprétation, les indices servent à accéder aux données (ou représentants) qui se trouvent en certaines positions. Un indice permet d'accéder à plusieurs représentants de la même valeur. Dans la littérature, un conteneur sans ses données est appelé *shape* [41] (ou forme). L'originalité de nos objets est leur forme particulière définie par une relation entre deux ensembles : les indices et les positions (cf. figure 2.1(b)).

2.4.3 Définition mathématique

Un objet mis en forme détaché de ses positions est une collection de valeurs indicées. Une telle collection est parfois appelée *champ de données* [15, 50, 40]. Un objet mis en forme est



(a) L'objet (b) La forme

FIG. 2.1 – *Un objet mis en forme et sa forme.*

donc un champ de données associé à une forme.

Soit $I = \cup_{n\in\mathbb{N}}\mathbb{Z}^n$, l'ensemble de tous les nuplets d'entiers.

Définition 12 (Forme) Une *forme* σ est une fonction partielle : $I \rightharpoonup I$. L'ensemble $\text{img}(\sigma)$ est son ensemble d'indices. L'ensemble $\text{def}(\sigma)$ est son ensemble de positions. Tout indice z est lié à l'ensemble des positions l telles que $\sigma(l) = z$. Chaque position est liée à exactement un indice et un indice est lié à une ou plusieurs position(s). Soit S, l'ensemble de toutes les formes.

Définition 13 (Champ de données) Soit V, un ensemble de valeurs. Un champ de données v à valeurs dans V est une fonction partielle : $I \rightharpoonup V$. Soit $DF(V)$, l'ensemble de tous les champs de données à valeurs dans V.

Un objet mis en forme est un champ de données v «associé à» une forme σ. Cette association signifie que v associe à chacune de ses valeurs, un indice de σ : autrement dit $\text{def}(v) \subseteq \text{img}(\sigma)$. De façon à définir proprement cette association, un objet mis en forme est défini en utilisant une fonction partielle :

Définition 14 (Objet mis en forme) Soit V, un type de données. Un objet mis en forme à valeurs dans V est une fonction partielle : $S \rightharpoonup DF(V)$ constante, qui à toute forme σ de son domaine de définition, associe un même champ de données v tel que $\text{def}(v) \subseteq \text{img}(\sigma)$.

Opérations

Les opérations sur les objets mis en forme sont les extensions naturelles des opérations sur les objets PEI, excepté le calcul et la superposition que nous combinons pour définir une seule opération appelée *opération globale*.

Définition 15 (Réindiçage) Soit X, un objet mis en forme. Soit h, une fonction partielle de I dans I. Le réindiçage est l'opération, qui appliquée à X et h, résulte en l'objet mis en forme Y défini par : $Y(\sigma) = X(h \circ \sigma) \circ h$.

Le réindiçage modifie la forme d'un objet PEI tout en préservant les valeurs et leurs positions (cf. figure 2.2).

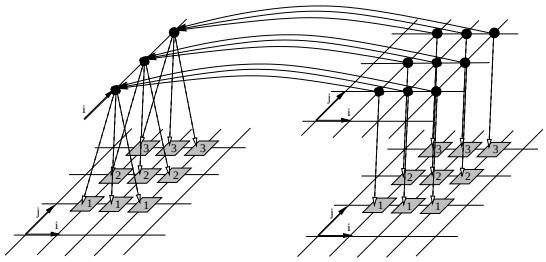


FIG. 2.2 – *Un exemple de réindiçage.*

Définition 16 (Déplacement) Soit X, un objet mis en forme. Soit g, une fonction partielle de I dans I. Le déplacement est l'opération, qui appliquée à X et g, résulte en l'objet mis en forme Y défini, pour toute forme σ telle que $\mathrm{def}(X(\sigma) \circ g) \subseteq \mathrm{img}(\sigma)$, par $Y(\sigma) = X(\sigma) \circ g$.

Le déplacement laisse la forme inchangée et modifie le champ de donnée en affectant d'autres indices à des valeurs dans le même ensemble de valeurs. Il induit un changement de la position des valeurs avec éventuelle copie ou suppression de valeurs.

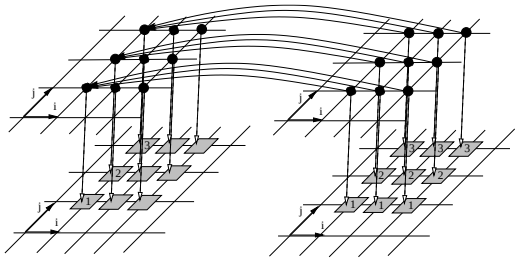


FIG. 2.3 – *Un exemple de déplacement.*

Définition 17 (Opération globale) Soit X et Y, deux objets mis en forme à valeurs dans V et W respectivement. Soit $\oplus$, une opération binaire (fonction totale) de $V \times W$ dans V'. L'opération globale est l'opération qui, appliquée à X et Y résulte en l'objet mis en forme Z défini par : $Z(\sigma) = X(\sigma) \bigoplus Y(\sigma)$ où $X(\sigma) \bigoplus Y(\sigma)$ est le champ de données de I dans V', défini par : $\mathrm{def}(X(\sigma) \bigoplus Y(\sigma)) = \mathrm{def}(X(\sigma)) \cap \mathrm{def}(Y(\sigma))$ et $(X(\sigma) \bigoplus Y(\sigma))(z) = X(\sigma)(z) \oplus Y(\sigma)(z)$.

Définition 18 (Énoncé) Nous appelons *énoncé* tout système d'équations dont les variables sont des objets mis en forme.

Notation mathématique : Nous introduisons un minimum de notations mathématiques pour les objets conformes et leur opérations afin de formuler la sémantique des expressions de la localité des données dans les langages data-parallèles.

Les opérations globales sur les objets mis en forme sont notées de la même façon que les opérations arithmétiques et logiques sur les scalaires (ex : $A + B$).

De façon à simplifier les notations, l'application d'un réindiçage et l'application d'un déplacement sur un objet mis en forme X sont notées uniformément $X.f$, où f est une paire (h, g) de fonctions partielles : h définissant un réindiçage et g, un déplacement, et $X.f$ est l'application sur X du réindiçage suivi du déplacement.

Enfin, pour les fonctions partielles : id est l'identité de I dans I, $\lambda x.e(x)$ est la fonction qui à tout x associe l'expression $e(x)$ et $f_{\setminus D}$ est la restriction de la fonction f au domaine D.

2.5 Adéquation du domaine sémantique

Nous donnons quelques exemples pour montrer l'adéquation entre notre domaine sémantique et les sémantiques intuitives des expressions de la localité des données dans les langages data-parallèles.

Exemple 10 (Somme de vecteurs non-alignés)

Reprenons l'exemple 7 page 26 : la dépendance entre les tableaux `A(0:7)`, `B(0:7)` et `C(0:7)` est définie par $\forall i \in [0..7]$ (`C[`i`]` = `A[`i`]` + `B[`i`]`). Elle est exprimée `C = A + B` en HPF indépendamment du placement des valeurs. Dans notre domaine sémantique, elle est définie par l'énoncé :

$$C = A.id_1 + B.id_2$$

où $id_1 = (h_1, g_1)$ et $id_2 = (h_2, g_2)$ sont des paires de fonctions partielles dont la composition est égale à l'identité restreinte à l'ensemble [0..7]. Ces paires de fonctions définissent un placement particulier des valeurs des tableaux. Par exemple, le placement des tableaux dans le programme HPF page 26 revient à écrire les définitions suivantes :

$$\begin{array}{ll} h_1 = g_1^{-1} & g_1 = \lambda i.i{+}1_{\setminus D} \\ h_2 = g_2^{-1} & g_2 = \lambda i.2{\times}i_{\setminus D} \end{array}$$

où $D = [0..7]$. Ce placement est décrit figure 2.4.

◇

Le rapport entre notre domaine sémantique et les langages data-parallèles est assez évident puisque notre domaine sémantique définit des variables parallèles, des positions de données, des opérations globales et des communications. L'exemple qui suit utilise notre domaine sémantique pour expliquer un point délicat dans les langages data-paralèles : la sémantique des indices d'une variable parallèle.

Dans des langages comme HPF, les indices sont des indices de tableaux sans référence au placement des données. En conséquence, si un programme exprime qu'une valeur en un certain indice dépend d'une valeur à un autre indice, cela peut vouloir dire n'importe qu'elle relation entre les positions des données, ce qui implique ou non une communication : dans un tel langage, les communications sont cachées ou implicites. De plus, l'association entre indices et positions est quelconque : elle peut ne pas être injective, i.e. un indice de tableau

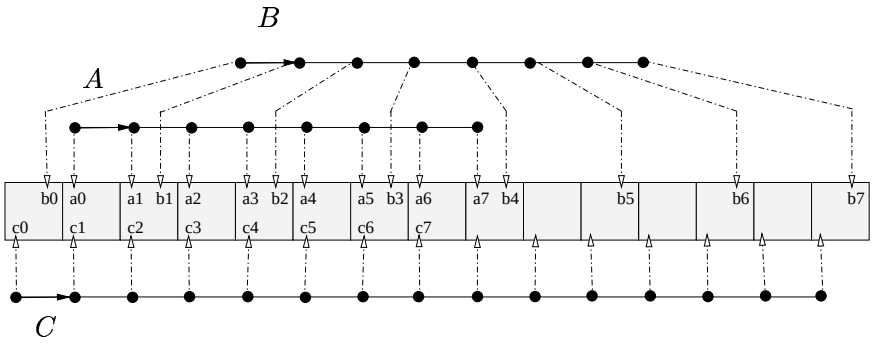


FIG. 2.4 – *Placements relatifs des valeurs des trois tableaux.*

peut correspondre à plusieurs positions. C'est un cas particulier d'alignement de données en utilisant des directives.

Dans d'autres langages comme C*, les indices référencent directement des processeurs virtuels : un indice de variable parallèle est un numéro de processeur virtuel et `X[`i`]` désigne une valeur locale au i-ième processeur virtuel. L'association entre indices et positions est donc injective et toute dépendance entre une valeur à un certain indice et une valeur à un autre, entraine une dépendance similaire entre les positions correspondantes. C'est pourquoi, les variables parallèles sont distribuées sur l'architecture physique et les communications sont explicites.

Exemple 11 (Produit de matrices)

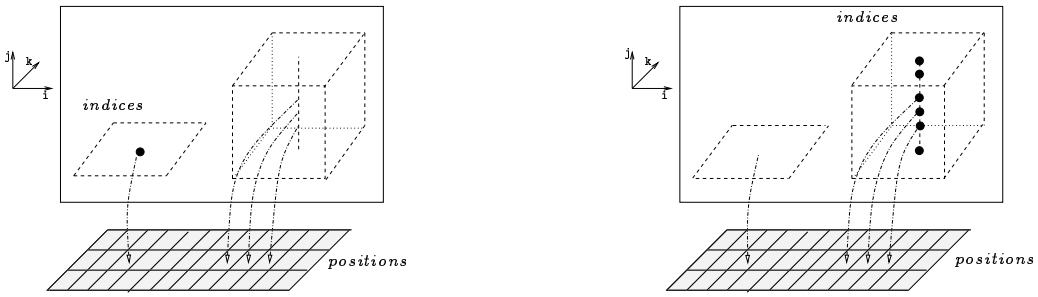
Considérons le calcul des produits `A[`i,k`]`×`B[`k,j`]`, pour tout $1 \leq i,j,k \leq n$, placés sur un tableau `P` tridimensionnel cubique. Voici la définition de ce problème dans notre domaine sémantique où l'on utilise des déplacements :

$$P = A.spread_1 \times B.spread_2$$

où $spread_1 = (h, g_1)$ et $spread_2 = (h, g_2)$, et
- $h = id$,
- $g_1 = \lambda(i,j,k).(i,k)_{\setminus D}$ et $g_2 = \lambda(i,j,k).(k,j)_{\setminus D}$.

où $D = [1..n] \times [1..n] \times [1..n]$.

Les objets mis en forme A et $A.spread_1$ ont la même forme : cette forme associe les indices de la matrice et du cube à des positions.
De plus, puisque des indices différents ne peuvent pas être associés à la même position de la forme, un indice du cube ne peut pas être associé à la même position qu'un indice de la matrice. En conséquence, la forme de ces objets mis en forme peut être une correspondance injective entre indices et positions. Ces explications sont illustrées par la figure 2.5. Cet énoncé peut donc être interprété dans les différentes sémantiques intuitives des langages data-parallèles, par exemple celle de C*.



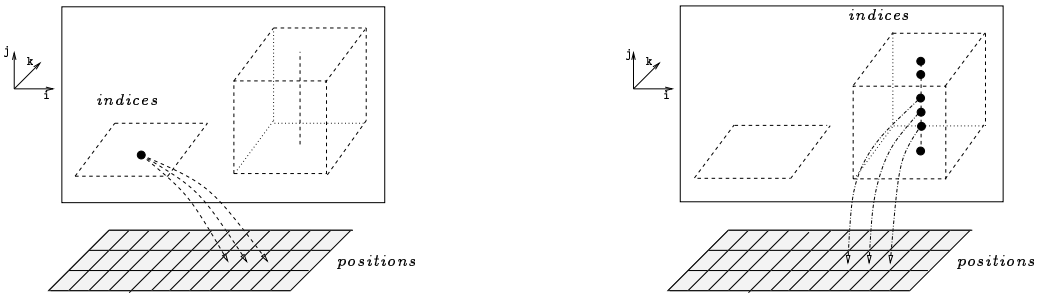
(a) L'objet A (b) L'objet $A.spread_1$

FIG. 2.5 – *Deux objets de même forme.*

Considérons maintenant un autre énoncé pour le même problème en utilisant un réindiçage à la place d'un déplacement :

$$P = A.spread_3 \times B.spread_4$$

où $spread_3 = (h_3, g)$ et $spread_4 = (h_4, g)$, et
- $g = id$,
- $h_3 = g_1$ et $h_4 = g_2$.



(a) L'objet A (b) L'objet $A.spread_3$

FIG. 2.6 – *Deux objets de formes différentes.*

Les objets mis en forme A et $A.spread_3$ ont deux formes différentes mais les mêmes valeurs associées aux mêmes positions. Puisque chaque indice sur une ligne verticale du cube (cf. figure 2.6) est lié à au moins une position et que les valeurs accédées par un indice de cette ligne sont des valeurs accédées par un unique indice dans A, alors cela signifie que tout indice de A est forcément lié à toutes les positions qui sont associés aux indices sur la ligne. Cela implique que la forme de A ne peut pas être une correspondance injective entre indices et positions.

Cet énoncé définit donc des indices virtuels dans un tableau qui sont associés à différentes positions sur une machine. C'est ce qui exprimé par un alignement de données en HPF en uti-

lisant une directive `ALIGN`. L'énoncé spécifie un algorithme différent puisque les dépendances, qui entraînent des communications dans l'énoncé précédent, ont été modifiées : elles concernent maintenant les positions et entraînent un alignement des données.
◇

2.6 Un bon équilibre

Nous venons de voir que notre domaine sémantique offre une notion de localité qui fait la jonction entre celle de HPF et celle de C*.

Nous montrons ici qu'il définit un bon degré de coopération entre le programmeur et le compilateur afin obtenir une implantation efficace. Le raisonnement pour obtenir un énoncé correspondant à une implantation efficace s'appuie sur un système de preuve et une sémantique opérationnelle associés aux énoncés considérés. Le système de preuve permet de prouver la correction des énoncés, tandis que la sémantique opérationnelle permet de mesurer son efficacité. Ces différents éclairages de notre domaine sémantique sont associés à un sous-ensemble des énoncés que nous appelons *énoncés bien formés*.

Les énoncés bien formés, le système de preuve ainsi que la sémantique opérationnelle ont été formellement définis dans [36, 67]. Nous en rappelons les principes essentiels ici. Un énoncé bien formé est un énoncé à assignation unique et tel que la forme de chacune des variables peut être inférée à partir de la forme d'une seule d'entre elles, appelée *template*. Étant donné un énoncé bien formé, une procédure automatisable détermine une forme pour chaque variable, de sorte à déterminer pour chaque variable, un domaine d'indices, un domaine de positions et les positions de chaque valeur. Ainsi, chaque équation d'un énoncé peut être vue soit comme une relation entre champs de données utilisée par le système de preuve, soit comme une règle de transition faisant partie de la sémantique opérationnelle. En particulier, voici ce que définit la sémantique opérationnelle :

- l'espace de processeurs virtuels,
- la position des données sur les processeurs virtuels,
- les calculs à réaliser par chacun des processeurs virtuels,
- les communications (virtuelles) entre processeurs virtuels,
- l'ordre dans lequel les calculs doivent être réalisés.

Ces concepts opérationnels très abstraits sont suffisants pour refléter une certaine notion d'efficacité en terme de communications et d'usage mémoire et sont conçus pour permetre au programmeur de choisir une transformation plutôt qu'une autre. En cela, notre objectif est différent du modèle BSP introduit par Valiant [62] qui vise à permettre au programmeur de prédire la performance de son programme sur une large variété d'architectures.
Nous reprenons l'exemple du produit de matrice pour montrer que notre domaine sémantique aide à résoudre les difficultés rencontrées par le programmeur et le compilateur en HPF et en C*. Un autre exemple est donnée dans [36] est conduit

Exemple 12 (Produit de matrices)

Nous définissons un premier énoncé pour ce problème, à partir du système d'équations récurrentes suivant :

$$\begin{array}{lcll} t_{i,j,-1} & = & 0 & \text{pour } 0 \leq i,j \leq n{-}1 \\ t_{i,j,k} & = & a_{i,k} \times b_{k,j} + t_{i,j,k-1} & \text{pour } 0 \leq i,j,k \leq n{-}1 \\ t_{i,j,n-1} & = & c_{i,j} & \text{pour } 0 \leq i,j \leq n{-}1 \end{array}$$

où les variables indicées a, b et c identifient les matrices $n \times n$, et t est une variable intermédiaire indicée sur $[0..n{-}1] \times [0..n{-}1] \times [-1..n{-}1]$.

Voici notre premier énoncé :

$$\begin{array}{lcl} T.init & = & 0 \\ T.current & = & A.spread_a \times B.spread_b + T.prec \\ T.term & = & C.expand \end{array}$$

- où A, B, C et T correspondent aux variables du système d'équations récurrentes précédent, et *init*, *current* et *term* définissent des déplacements qui partitionnent T en trois parties. Ils sont définis ainsi :

$$\begin{array}{lcl} init & = & (id, \lambda(i,j,k).(i,j,k)_{\setminus D_{-1}}) \\ current & = & (id, \lambda(i,j,k).(i,j,k)_{\setminus D}) \\ term & = & (id, \lambda(i,j,k).(i,j,k)_{\setminus D_{n-1}}) \end{array}$$

  avec :

$$\begin{array}{l} D_{-1} = [0..n-1] \times [0..n-1] \times \{-1\} \\ D = [0..n-1] \times [0..n-1] \times [0..n-1] \\ D_{n-1} = [0..n-1] \times [0..n-1] \times \{n-1\} \end{array}$$

- où $spread_a = (h_a, g_a)$, $spread_b = (h_b, g_b)$, $prec = (h_p, g_p)$ et $expand = (h_e, g_e)$ définissent les dépendances du système d'équations récurrentes et sont tels que :

$$\begin{array}{lcl} h_a \circ g_a & = & \lambda(i,j,k).(i,k)_{\setminus D} \\ h_b \circ g_b & = & \lambda(i,j,k).(k,j)_{\setminus D} \\ h_p \circ g_p & = & \lambda(i,j,k).(i,j,k{-}1)_{\setminus D} \\ h_e \circ g_e & = & \lambda(i,j,k).(i,j)_{\setminus D_{n-1}} \end{array}$$

De telles dépendances déterminent le cadencement des calculs. Sans d'autres précisions, la sémantique de l'énoncé précédent pourrait être celle du programme suivant qui reprend la syntaxe de HPF :

```
REAL A(0:N-1,0:N-1),B(0:N-1,0:N-1),C(0:N-1,0:N-1)
REAL T(0:N-1,0:N-1,-1:N-1)
...
FORALL (I=0:N-1,J=0:N-1)
  DO K=0,N-1
    T(I,J,K)=A(I,K)*B(K,J)+T(I,J,K-1)
  END DO
END FORALL
FORALL (I=0:N-1,J=0:N-1)
    C(I,J)=T(I,J,N-1)
END FORALL
...
```

Ce programme explicite l'ordre partiel des calculs, mais laisse indéterminée la relation entre indices de tableaux et processeurs virtuels. Notons que ce n'est pas un programme HPF syntaxiquement correct puisqu'il est interdit de placer une boucle `DO` à l'intérieur d'un `FORALL` dans un programme HPF. Cette interdiction est évidemment liée aux difficultés rencontrées par le compilateur.

Instancier (h_a, g_a), (h_b, g_b), (h_p, g_p) et (h_e, g_e) en choisissant des paires de fonctions particulières a pour effet d'attacher à l'énoncé des propriétés opérationnelles supplémentaires, tout en préservant sa correction. Cette sorte de raffinement opérationnel est tout à fait comparable à l'ajout d'une directive d'alignement dans un programme HPF, mais en considérant les directives obligatoirement respectées par le compilateur. Nous reviendrons sur cette notion de directives dans la section suivante entièrement dédiée.

Un énoncé tel que le précédent peut être transformé étape par étape soit par le programmeur, soit par le compilateur, par exemple pour définir un meilleur compromis entre volume de communications et usage mémoire. Certaines de ces transformations peuvent consister en la substitution d'une paire de fonctions, disons (h', g'), à une autre paire, disons (h, g), à condition que $h' \circ g' = h \circ g$. L'énoncé ainsi transformé exprime toujours les mêmes dépendances, donc demeure correct, mais une relation différente entre les indices et les positions, donc est attaché à une signification opérationnelle différente.

Par exemple, considérons les cas suivants où nous étudions la position des valeurs de la matrice a relativement à celles de t. Dans chacun des cas, le placement des données est déterminé par une instance particulière de (h_a, g_a).

Nous illustrons chaque cas par une figure qui montre les positions de l'objet T. De façon à simplifier la figure, une position de T est confondue avec l'indice auquel elle est liée par la forme de T. L'ensemble des positions de A qui est un sous-ensemble des positions de T, est représenté en grisé. Enfin, une flèche entre deux positions représente une communication (virtuelle).

1. Répliquation (Fig.2.7). En choisissant ces définitions :

$$h_a = \lambda(i,j,k).(i,k)_{\setminus D}$$
$$g_a = id$$

Nous exprimons que la valeur $a_{i,k}$ est située en toutes les positions où se situe une valeur quelconque parmi les $t_{i,j,k}$, $j \in [0..n-1]$. En conséquence, il n'est pas nécessaire de communiquer les valeurs de la matrice a pour calculer les produits. En contrepartie chaque valeur $a_{i,k}$ est (virtuellement) stockée n fois.

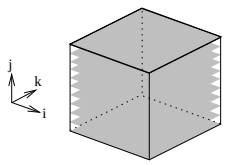


FIG. 2.7 – *Répliquation de matrice.*

2. Diffusion (Fig.2.8). En choisissant ces définitions :

$$h_a = id$$
$$g_a = \lambda(i,j,k).(i,k)_{\setminus D}$$

Nous exprimons que la valeur $a_{i,k}$ a sa position propre, différente des positions où les valeurs de t sont situées. En conséquence, chaque valeur $a_{i,k}$ est diffusée en toutes les positions où une quelconque valeur $t_{i,j,k}$, $j \in [0..n-1]$ se situe.

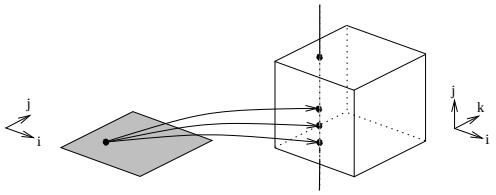


FIG. 2.8 – *Diffusion de matrice.*

3. Alignement et diffusion (Fig.2.9). En choisissant ces définitions :

$$h_a = \lambda(i,j,k).(i,k)_{\setminus [0..n-1]\times\{0\}\times[0..n-1]}$$
$$g_a = \lambda(i,j,k).(i,0,k)_{\setminus D}$$

Nous exprimons que la valeur $a_{i,k}$ est placée à la même position que $t_{i,0,k}$. Comme dans le cas précédent, la valeur $a_{i,k}$ est diffusée en toutes les positions où une valeur quelconque $t_{i,j,k}$, $j \in [0..n-1]$ se situe.

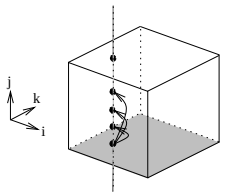


FIG. 2.9 – *Alignement et diffusion de matrice.*

4. Algorithme de Cannon's (Fig.2.11).

Les configurations précédentes peuvent ne pas être suffisamment prescriptives pour le compilateur, par exemple si la diffusion ne peut pas être implantée efficacement en raison des spécificités de l'architecture. Dans de tels cas, le programmeur peut être d'une aide précieuse en indiquant au compilateur qu'il peut changer l'ordre dans lequel les produits sont calculés et accumulés. Le compilateur peut exploiter cette indication pour trouver un meilleur compromis entre communications et place mémoire. Cette indication pourrait être donnée en utilisant une notation proche de la syntaxe HPF, en insérant une directive `DO INDEPENDENT` dans le programme précédent.

La sémantique du nouveau programme est donnée par l'énoncé qui comporte ces définitions :

$$h_a \circ g_a = \lambda(i,j,k).(i, h_{i,j}(k))_{\setminus D}$$
$$h_b \circ g_b = \lambda(i,j,k).(h_{i,j}(k), j)_{\setminus D}$$

où $h_{i,j}$, $i,j \in [0..n{-}1]$, est une permutation quelconque de $[0..n{-}1]$.

Considérons à nouveau le placement de A. Comme nous l'avons vu précédemment, il est possible de séparer les dépendances en un alignement et une diffusion.

Un cas intéressant est le cas où la dépendance autorise les valeurs de a à être d'abord alignées avec les valeurs $t_{i,j,0}$, puis diffusées le long de l'axe k, comme le montre la figure 2.10. Dans ce cas, la diffusion peut être réalisée en même temps que le somme

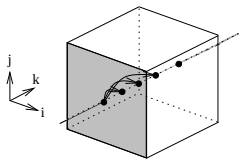


FIG. 2.10 – *Alignement et diffusion de matrice selon un certain axe.*

des produits. Un tel alignement est permis ssi $\lambda(i,j).(i, h_{i,j}(0))$ et $\lambda(i,j).(h_{i,j}(0), j)$ sont deux permutations de $[0..n{-}1]\times[0..n{-}1]$. Ce raisonnement peut être conduit soit par le programmeur, soit le le compilateur et peut mener à la définition suivante pour $h_{i,j}$:

$$h_{i,j}(k) = (i{+}j{-}k)\%n$$

Cette définition induit un énoncé équivalent où la paire (h_a, g_a) est définie comme suit :

$$h_a = \lambda(i,j,k).(i, (i{+}j)\%n)_{\setminus [0..n-1]\times[0..n-1]\times\{0\}}$$
$$g_a = \lambda(i,j,k).(i, (j{-}k)\%n, 0)_{\setminus D}$$

La fonction h_a définit le placement initial de l'algorithme de Cannon, tandis que g_a exprime les communications à réaliser.

A partir de cet énoncé, le compilateur peut utiliser une technique classique appelée *uniformisation* pour rendre les communications uniformes (Fig.2.11) et implanter ces communications en utilisant les liens d'une architecture parallèle régulière.

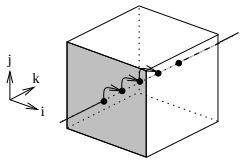


FIG. 2.11 – *Communications uniformes.*

Appliquer les même transformations à l'objet B, c'est-à-dire définir (h_b, g_b) comme suit :

$$h_b = \lambda(i,j,k).((i+j)\%n, j)_{\setminus [0..n-1]\times[0..n-1]\times\{0\}}$$
$$g_b = \lambda(i,j,k).((i-k)\%n, j, 0)_{\setminus D}$$

mène à l'algorithme de Cannon.

◇

2.7 De l'utilisation des directives comme outil de coopération

Une spécificité des langages parallèles industriels actuels HPF et OPENMP est la définition de directives. Ces directives sont un indéniable confort pour le programmeur qui lui donne la possibilité de préciser incrémentalement la sémantique opérationnelle de son programme tout en préservant sa correction (sauf pour certaines directives OPENMP). L'utilisation de directives est certainement une approche prometteuse pour une programmation parallèle de haut niveau. Toutefois elle pose certaines questions sémantiques : en particulier un problème se pose lorsque les directives ne sont pas obligatoirement respectées par le compilateur. C'est le cas en HPF où il est donc difficile de connaître la sémantique opérationnelle réelle du programme.

L'objet de la thèse de Philippe Gerner [35] est l'étude théorique de la notion de directive afin d'éclaircir leur sémantique et dans le but de se servir des directives comme un véritable outil de programmation.

Cette étude vise à montrer comment il est possible d'établir formellement la sémantique des directives. Elle est appliquée au cas du data-parallélisme, et en particulier au langage industriel HPF. Elle conduit à distinguer deux sémantiques opérationnelles pour un langage : la sémantique *officielle*, qui sert à expliquer le langage et la sémantique *effective*, qui définit ce à quoi est tenu un compilateur du langage. La sémantique effective est importante dans cette étude car c'est elle qui donne un sens aux directives. La somme des deux sémantiques est appelée *sémantique de référence* et constitue une norme du langage auquelle programmeur et compilateur peuvent se référer.

Philippe Gerner propose une méthode pour décrire formellement la sémantique opérationnelle des langages. L'introduction de cette méthode spécifique est motivée par le fait que parmi les méthodes existantes exposées dans l'état de l'art, aucune n'est conçue pour exposer conjointement plusieurs sémantiques pour un même langage, et les relier entre elles. Dans cette méthode, chacune des sémantiques (officielle, effective) associe un graphe d'exécution à tout programme. Un compilateur doit respecter la sémantique effective, mais peut contraindre cette sémantique afin de l'adapter au mieux à l'architecture cible. Ceci correspond à un *raffinement* de la sémantique effective. Cette notion de raffinement est formellement définie comme l'ajout d'états intermédiaires dans les graphes d'exécution correspondant à la sémantique effective des programmes.

La méthode est appliquée pour donner une sémantique aux directives. Cette sémantique est une relation d'ordre qui exprime la *préférence* de l'utilisateur du langage quant au respect ou non d'une directive par le compilateur.

Ces travaux montrent comment pourrait être construit un langage à directives établissant un contrat clair entre programmeur et compilateur. Un tel contrat permet de concevoir un

langage qui «prend de l'avance» sur les capacités du compilateur à traiter les directives. Du point de vue de la coopération programmeur-compilateur, il est évidemment souhaitable que la plupart des directives puissent être traitées par le compilateur. Autrement dit, le contrat ne doit pas être trop en défaveur du compilateur.

2.8 Conclusion

La définition du data-parallélisme doit inclure une notion de localité des données où les notions intuitives différentes des principaux langages data-parallèles standards HPF et C* peuvent se rejoindre.

Nous avons proposé un domaine sémantique, issu de PEI, définissant une notion de localité qui fait la jonction entre les notions de shape en C* et d'alignement en HPF.

Nous avons conçu ce domaine sémantique dans l'optique de définir un médium idéal entre le programmeur et le compilateur établissant un meilleur partage du travail en vue d'obtenir un programme parallèle efficace. En comparaison avec la notion d'alignement en HPF, le travail du compilateur est moins important car un énoncé à notre sens définit plus précisément le placement des données. En comparaison avec la notion de shape en C*, le travail du programmeur est simplifié car un énoncé peut être plus facilement transformé.

Nous pensons que ce domaine sémantique peut aider à la fois le programmeur et le compilateur à transformer le programme de manière à atteindre des implantations efficaces.

Chapitre 3

Applications et perspectives

3.1 Introduction

Les travaux qui sont présentés dans ce chapitre concernent d'une part l'implantation sur architectures parallèles des méthodes numériques de résolution d'équations aux dérivées partielles (EDP) et d'autre part l'utilisation des grilles de calcul.

Ces recherches constituent des prolongements concrets de mes travaux antérieurs dans le domaine de la construction de programmes data-parallèles par des techniques de transformations. J'ai mené ces recherches avec l'objectif d'appliquer mes résultats théoriques.

Mes recherches sur l'implantation des méthodes numériques ont été effectuées en collaboration avec Éric Sonnendrücker, Professeur à l'ULP, membre de l'IRMA (Institut de Recherche Mathématique Avancé, UMR 7501 ULP-CNRS) et Francis Filbet, chargé de recherche CNRS, tous deux mathématiciens appliqués. Ces travaux sont poursuivis dans le cadre du projet INRIA CALVI [2]. Mes recherches sur l'utilisation des grilles de calcul s'inscrivent dans le cadre du projet TAG [1] mis en place dans l'équipe ICPS du laboratoire LSIIT à Strasbourg.

La résolution numérique d'EDP constitue un domaine d'application important du parallélisme. Les EDP tiennent une place de plus en plus grande dans la modélisation de systèmes physiques. La simulation numérique de ces systèmes permet des avancées en sciences physiques avec des implications importantes dans le futur. Bien sûr, obtenir des simulations réalistes et en temps interactif suppose d'utiliser efficacement les ressources de calcul disponibles.

Beaucoup de recherches sont à mener dans ce domaine pour permettre de concilier les méthodes numériques de résolution avec les techniques de parallélisation de façon à concevoir des algorithmes parallèles qui utilisent les ressources disponibles de manière optimale.

La simulation numérique de certains systèmes physiques présente une grande difficulté du fait de la grande taille du système et le grand nombre de dimensions de l'espace physique considéré. L'obtention d'une approximation suffisante de la solution requiert alors des ressources de calcul considérables que seule une *grille de calcul* peut offrir.

Les progrès en matière d'uniformisation et de transparence d'accès aux composants permettent d'agréger des machines hétérogènes en les connectant par un réseau lui-même hétérogène. L'utilisation de tels agrégats, nommés grilles de calcul, est rendu possible par l'émergence d'infrastructures logicielles permettant d'exécuter un code sur une telle architecture (par

exemple GLOBUS [31] interfacé avec MPI [30]). Les grilles de calcul représentent un potentiel de calcul inépuisable [29].

Cependant, beaucoup de progrès restent à faire pour les exploiter efficacement, particulièrement en matière de programmation. Les codes parallèles conçus pour une machine parallèle homogène sont peu performants lorsqu'on les exécute sur une grille de calcul. A cela, il y a principalement deux raisons : la première est que les processeurs de la grille sont hétérogènes et la charge de travail allouée aux processeurs est donc souvent déséquilibrée, la deuxième est que les liens réseaux sont de plusieurs ordres plus lents sur une grille que sur une machine parallèle. L'équilibrage de charge est un objectif majeur dans ce contexte. Cependant, peu de recherches concernent des stratégies d'équilibrage de charge spécifiquement guidées par le code à exécuter. Le projet AppLeS [9] utilise des informations au sujet du code afin d'ordonnancer l'exécution des processus, mais il ne modifie pas le code source pour améliorer son exécution.

Nos résultats sont basés sur l'étude d'une application en physique des plasmas. Cette application est un solveur de l'équation de Vlasov par la méthode PFC [26].

Une partie de ces résultats concernent l'implantation de cette méthode sur une architecture parallèle homogène. Cette méthode induit un algorithme parallèle simple avec des phases de calcul et de communication bien distinctes. Nous avons conçu un algorithme plus performant qui se déduit du premier algorithme en appliquant un changement de base qui optimise le recouvrement des communications par les calculs.

Une autre partie concerne l'utilisation des grilles de calcul. Nous avons défini une transformation de code applicable à une classe étendue de codes MPI conçus pour une machine parallèle homogène. Cette transformation permet d'obtenir un code plus adapté à la grille de calcul. Elle vise à équilibrer la charge sur les processeurs hétérogènes de la grille.

Ce chapitre présente ces différents travaux en commençant par une présentation de l'application qui nous a servi à valider les transformations que nous avons défini.

3.2 Une application en physique des plasmas

Notre application est un solveur de l'équation de Vlasov. La résolution de l'équation de Vlasov est habituellement réalisée par des méthodes numériques particulaires (Particle-In-Cell) qui approximent le plasma par un nombre fini de particules. Ces méthodes mènent à des résultats satisfaisants avec un relativement petit nombre de particules. Cependant, il est bien connu que le bruit numérique qui accompagne une méthode particulaire devient dans certains cas trop important pour obtenir une description fiable de la fonction de distribution.

La méthode de résolution utilisée dans notre application est une méthode non particulaire, nommée PFC (Positive and Flux Conservative method) [26]. Une autre méthode non-particulaire est la méthode semi-Lagrangienne [25]. Les méthodes non particulaires discrétisent l'équation de Vlasov sur l'*espace des phases*, i.e. l'espace des positions et des vitesses. De telles méthodes sont d'un grand intérêt pour obtenir une bonne description de l'évolution du plasma ou du faisceau de particules. En particulier, la méthode PFC garantit la conservation de certaines caractéristiques de la *fonction de distribution*. Cependant, ces méthodes impliquent un grand volume de données et de calculs, parce que l'inconnue est calculée en tout point de l'espace des phases.

3.2.1 Équation de Vlasov - Schéma de résolution

L'ensemble de particules est décrit par une *densité de particules* $f(t, \mathrm{x}, \mathrm{v})$ qui dépend de l'instant t, de la position x et de la vitesse v. La fonction f, appelée *fonction de distribution*, est l'inconnue de l'équation de Vlasov. L'évolution de la densité de particules $f(t, \mathrm{x}, \mathrm{v})d\mathrm{x}\, d\mathrm{v}$ dans l'espace des phases, $(\mathrm{x}, \mathrm{v}) \in \mathbb{R}^d \times \mathbb{R}^d$, $d = 1, .., 3$, est donnée par l'équation de Vlasov normalisée :

$$\frac{\partial f}{\partial t} + \mathrm{div}_\mathrm{x}(\mathrm{v}\, f) + \mathrm{div}_\mathrm{v}(E(t, \mathrm{x})\, f) = 0. \tag{3.1}$$

où le *champ électrique* E est défini par (en utilisant l'équation de Poisson) :

$$E(t, \mathrm{x}) = -\nabla_\mathrm{x}\phi(t, \mathrm{x}), \quad -\Delta_\mathrm{x}\phi(t, \mathrm{x}) = \rho(t, \mathrm{x}), \tag{3.2}$$

où la *densité de charge* ρ est définie par :

$$\rho(t, \mathrm{x}) = \int_{\mathbb{R}^d} f(t, \mathrm{x}, \mathrm{v}) d\mathrm{v}. \tag{3.3}$$

3.2.2 La méthode PFC

La méthode PFC ainsi que la méthode semi-Lagrangienne résolvent une équation différentielle de la forme :

$$\frac{\partial f}{\partial t} + \mathrm{div}_\mathrm{x}(U(t, \mathrm{x})\, f) = 0 \tag{3.4}$$

Ces méthodes peuvent s'appliquer à la résolution de l'équation de Vlasov après une discrétisation dans le temps basée sur le découpage de l'équation de Vlasov en deux équations de la forme (3.4). Les méthodes semi-Lagrangienne et PFC ne diffèrent que dans la façon de résoudre ces équations. Les équations obtenues après découpage de l'équation de Vlasov sont des équations de transport également appelées advections, en x et en v. La discrétisation dans le temps résulte en la procédure suivante sur $\Delta t = [t^n, t^{n+1}]$ connaissant une solution approchée f^n à l'instant t^n.

1. Résoudre sur Δt l'advection en x :

$$\begin{cases} \dfrac{\partial f^{(1)}}{\partial t} + \mathrm{div}_\mathrm{x}\left(\mathrm{v}\, f^{(1)}\right) = 0, \\ \\ f^{(1)}(0, \mathrm{x}, \mathrm{v}) = f^n(\mathrm{x}, \mathrm{v}). \end{cases} \tag{3.5}$$

2. Calculer le champ électrique $E(t^{n+1/2}, \mathrm{x})$ à l'instant $t^{n+1/2}$ en substituant $f^{(1)}(\Delta t, \mathrm{x}, \mathrm{v})$ dans l'équation de Poisson et dans (3.3).
3. Résoudre sur Δt l'advection en v :

$$\begin{cases} \dfrac{\partial f^{(2)}}{\partial t} + \mathrm{div}_\mathrm{v}\left(E(t^{n+1/2}, \mathrm{x})\, f^{(2)}\right) = 0, \\ \\ f^{(2)}(0, \mathrm{x}, \mathrm{v}) = f^{(1)}(\Delta t, \mathrm{x}, \mathrm{v}). \end{cases} \tag{3.6}$$

et poser $f(t^{n+1}, \mathrm{x}, \mathrm{v}) = f^{(2)}(\Delta t, \mathrm{x}, \mathrm{v})$.

Cette procédure nous ramène à la résolution des équations (3.5) et (3.6) sur un maillage de l'espace des phases. Considérons le cas $d = 2$, i.e. $(\mathrm{x}, \mathrm{v}) \in \mathbb{R}^4$ et introduisons un ensemble fini de points de l'espace des phases $(\mathrm{x}_i = (x_i, y_i))_{i \in \{0,..,n_\mathrm{x}\}}$ et $(\mathrm{v}_j = (v_{xj}, v_{yj}))_{j \in \{0,..,n_\mathrm{v}\}}$. Notons $\Delta\mathrm{x} = x_{i+1} - x_i = y_{i+1} - y_i$ le pas de position et $\Delta\mathrm{v} = v_{xj+1} - v_{xj} = v_{yj+1} - v_{yj}$ le pas de vitesse. En supposant que les valeurs de la fonction de distribution f, stockées dans la matrice $(F^n_{i,j})_{i,j}$, sont connues à l'instant $t^n = n\,\Delta t$. Nous trouvons les nouvelles valeurs de f à l'instant t^{n+1} en résolvant (3.5) et (3.6) sur chaque volume $[\mathrm{x}_i, \mathrm{x}_{i+1}] \times [\mathrm{v}_j, \mathrm{v}_{j+1}]$ de l'instant t^n à l'instant t^{n+1}. En utilisant le schéma PFC donné en [26], nous obtenons une approximation de l'équation (3.5) dont la solution $F^{(1)}_{i,j}$ est calculée à partir des valeurs de $F^n_{i,j}$ en utilisant un opérateur discret $Q^{(1)}_{\Delta\mathrm{x},\Delta\mathrm{v}}$:

$$F^{(1)}_{i,j} = [Q^{(1)}_{\Delta\mathrm{x},\Delta\mathrm{v}}(F^n_{0,j}, F^n_{1,j}, .., F^n_{n_\mathrm{x}-1,j})]_i. \tag{3.7}$$

Notons que la valeur $F^{(1)}_{i,j}$ ne dépend que des valeurs sur la ligne i de F^n. A partir de ces nouvelles valeurs, nous approximons le champ électrique sur le maillage de l'espace des positions $(\mathrm{x}_i)_{i \in \{0,..,n_\mathrm{x}\}}$ à partir de la charge discrète $\rho^{n+1/2}_i = \Delta\mathrm{v} \sum_j F^{(1)}_{i,j}$ et en utilisant une transformée de Fourier (FFT). Finalement, nous obtenons la solution à l'instant t^{n+1} en approximant l'équation (3.6) :

$$F^{n+1}_{i,j} = [Q^{(2)}_{\Delta\mathrm{x},\Delta\mathrm{v}}(F^{(1)}_{i,0}, F^{(1)}_{i,1}, .., F^{(1)}_{i,n_\mathrm{v}-1})]_j. \tag{3.8}$$

On remarque que la valeur $F^{n+1}_{i,j}$ ne dépend que des valeurs sur la colonne j de $F^{(1)}$.

3.2.3 L'algorithme induit

Lorsque les valeurs de la fonction de distribution sont stockées dans une matrice $(F^n_{i,j})_{i,j}$, où i et j représentent une position et une vitesse, le découpage de l'équation de Vlasov en deux advections introduit deux phases de calcul des éléments de la matrice : durant la première phase, chaque élément ne dépend que des éléments sur la même ligne et durant la seconde phase, chaque élément ne dépend que des éléments sur la même colonne.

Cette propriété induit un algorithme parallèle performant décrit dans [20] pour la méthode semi-Lagrangienne et dans [26] pour la méthode PFC. Bien entendu, l'obtention d'algorithmes parallèles plus efficaces requiert une analyse plus fine des dépendances : par exemple, pour la méthode PFC, cela signifierait de prendre en compte la définition des opérateurs $Q^{(1)}_{\Delta\mathrm{x},\Delta\mathrm{v}}$ et $Q^{(2)}_{\Delta\mathrm{x},\Delta\mathrm{v}}$ propres à cette méthode. L'intérêt de cet algorithme tient en sa généricité et sa simplicité. Nous rappelons cet algorithme ci-après pour la méthode PFC. Il décrit des opérations globales impliquant tous les processeurs utilisés :

Algorithme 1 :

Initialement la matrice F^n est distribuée par blocs de lignes.
Pour chaque étape de temps Δt

1. Calculer la matrice $F^{(1)}$ en utilisant l'opérateur $Q^{(1)}$ à partir des valeurs de F^n.
   (Cette phase ne requiert aucune communication)

2. Redistribuer la matrice $F^{(1)}$ de façon à obtenir une matrice $(F^{(1)^T})$ distribuée par blocs de colonnes.
3. Sommer les colonnes de $F^{(1)^T}$ pour obtenir la charge discrète ρ et calculer le champ électrique E.
   (Sans communication)
4. Calculer la matrice F^{n+1^T} en utilisant l'opérateur $Q^{(2)}$ à partir des valeurs de $F^{(1)^T}$.
   (Cette phase ne requiert aucune communication)
5. Redistribuer la matrice F^{n+1^T} de façon à obtenir une matrice (F^{n+1}) distribuée par blocs de lignes, avant d'aborder l'étape suivante.

Cet algorithme a été implanté par Francis Filbet en C++ avec appels à MPI. Le nom du code est VADOR.

3.3 Recouvrement des communications par les calculs

L'algorithme 1 décrit précédemment contient des phases de calcul sans communication et des phases de communication sans calcul. Les phases de communication de l'algorithme 1 consistent en une transposition par bloc c'est-à-dire un échange global des données qui est une opération coûteuse. Il est nécessaire d'améliorer cet algorithme.

Une approche pour améliorer le code consiste à optimiser les communications, par exemple, dans [59, 16], les auteurs décrivent une implantation efficace de la transposition. Nous choisissons une autre approche qui consiste à modifier l'algorithme de façon à exploiter une propriété de l'algorithme 1 : la séparation des calculs et des communications en phases bien distinctes. Il devient clair que le seul mécanisme qui peut être utilisé pour améliorer le code de manière significative est le recouvrement des communications par des calculs [19]. Ce mécanisme peut être utilisé durant les étapes (1)-(2) et (4)-(5) de l'algorithme 1 afin que des données soient calculées pendant que d'autres s'échangent.

Considérons les étapes (1)-(2) et une subdivision de la matrice $F^{(1)}$ en $p \times p$ blocs $(B_{k,l})_{k,l}$ de même taille où p est le nombre de processeurs utilisés. Le processeur k calcule d'abord les blocs $(B_{k,l})_{k,l}$, $l = 0, .., p-1$, puis envoie le bloc $(B_{k,l})_{k,l}$ au processeur l, pour tout $l \in \{0, .., p-1\} \setminus \{k\}$. Ces tâches peuvent être effectuées dans n'importe quel ordre à la seule condition qu'un bloc soit calculé avant d'être envoyé. Le recouvrement dépend de l'ordre dans lequel ces tâches sont accomplies. Nous souhaitons obtenir un code qui implante un recouvrement optimal.

En utilisant nos développements théoriques et le cadre formel Pei , ce problème peut être modélisé comme suit : les blocs de la matrice sont des valeurs placées dans un domaine de référence qui est un carré $[0..p-1] \times [0..p-1]$ de $\mathbb{Z}^2$: chaque valeur est naturellement associée à un couple (k, l) formé des coordonnées d'un point de $\mathbb{Z}^2$. Le problème consiste à expliciter une bijection σ qui induit un ordre opérationnel approprié. En posant $\sigma(k, l) = (alloc(k, l), t(k, l))$ avec $alloc : [0..p-1] \times [0..p-1] \rightharpoonup [0..p-1]$ et $t : [0..p-1] \times [0..p-1] \rightharpoonup [0..p-1]$, et en choisissant l'ordre partiel $<$ tel que $\sigma(z) < \sigma(z')$ si $alloc(z) = alloc(z') \wedge t(z) < t(z')$, nous définissons de façon classique une allocation et un cadencement des calculs. Une analyse de dépendances, nous donne $alloc(k, l) = k$ et $t(k, l) = h_k(l)$ où h_k est une permutation quelconque de $[0..p-1]$.

Maximiser le recouvrement revient à maximiser le minimum des délais entre l'instant auquel la valeur d'indice (k,l) doit être reçue et l'instant auquel cette valeur peut être envoyée. Le problème revient donc à trouver les h_k qui maximisent $min\{\, p-h_k(l) \mid 0 \leq k,l \leq p-1 \wedge k \neq l \,\}$, soit à minimiser $max\{\, h_k(l) \mid 0 \leq k,l \leq p-1 \wedge k \neq l \,\}$. Les permutations h_k solutions du problème sont celles qui vérifient $h_k(l) = p-1$ pour $k = l$. Par exemple, $h_k(l) = (l-k+p-1)\%p$ est une solution du problème. A condition de modéliser le problème dans le cadre formel PEI, le raisonnement menant à cette solution peut être conduit aussi bien par le compilateur que par le programmeur. Il repose sur une transformation d'énoncé qui raffine opérationnellement l'énoncé initial : plus précisément, cette transformation explicite le cadencement des calculs. Cette transformation est un changement de base défini par le réindiçage déterminé par une bijection qui à (k,l) associe $(k,(l-k+p-1)\%p)$. L'énoncé raffiné correspond à un code parallèle que nous avons obtenu en réécrivant le code VADOR. Dans ce nouveau code, l'envoi d'un bloc commence juste après son calcul et le calcul du bloc $B_{k,k}$ est réalisé en dernier conformément à l'algorithme que voici :

Algorithme 2 :

Pour chaque étape de temps Δt ; pour chaque processeur k

1. Pour tout $l = 1,..,p-1$,
   - calculer le bloc $B_{k,(k+l)\%p}$ de la matrice $F^{(1)}$ en appliquant $Q^{(1)}$ à partir des valeurs de F^n.
   - envoyer le bloc $B_{k,(k+l)\%p}$ au processeur $(k+l)\%p$.
   (Cette communication sera recouverte par le calcul du prochain bloc $B_{k,(k+l+1)\%p}$).
   - initialiser la réception du bloc $B_{(k-l)\%p,k}$ provenant du processeur $(k-l)\%p$.

   fin.
   Calculer le bloc $B_{k,k}$ de la matrice $F^{(1)}$.

2. Les blocs reçus sont stockés dans la matrice ${F^{(1)}}^T$, qui est sommée pour obtenir la charge discrète ρ. Le champ électrique E est calculé à partir de ρ.

3. Pour tout $l = 1,..,p-1$,
   - calculer le bloc $B_{k,(k+l)\%p}$ de la matrice ${F^{n+1}}^T$ en appliquant $Q^{(2)}$ à partir des valeurs de ${F^{(1)}}^T$.
   - envoyer le bloc $B_{k,(k+l)\%p}$ au processeur $(k+l)\%p$.
   (Cette communication sera recouverte par le calcul du prochain bloc $B_{k,(k+l+1)\%p}$).
   - initialiser la réception du bloc $B_{(k-l)\%p,k}$ provenant du processeur $(k-l)\%p$.

   fin.
   Calculer le bloc $B_{k,k}$ de la matrice ${F^{n+1}}^T$.

Les résultats expérimentaux [71] montrent que l'algorithme 2 permet de réduire de manière significative les temps d'exécution du code VADOR.

3.4 Adaptation à une grille de calcul

Le code VADOR décrit un volume de données et de calcul considérable : il est donc un bon candidat pour les grilles de calcul. L'étude de ce code nous a permis de proposer une transformation de code pour adapter un code MPI conçu pour une machine parallèle à une grille de calcul.

Cette transformation est présentée dans [68, 23]. Elle a pour but d'équilibrer la charge de calcul sur les processeurs hétérogènes de la grille de calcul de façon à réduire le temps d'exécution total. Elle permet de modifier la répartition de la charge décrite par le code initial. Un intérêt de cette transformation est qu'elle est applicable à une large classe de codes parallèles.

Le principe de cette transformation est issue des idées suivantes : une solution naïve pour équilibrer la charge consiste à faire en sorte que le système exécute plusieurs processus sur les processeurs les plus rapides. Évidemment, cette solution entraîne un surcoût système dû aux mécanismes de temps partagé et à la gestion des communications entre processus. L'idée que nous avons suivie est de réécrire le code de façon à ce qu'un seul processus soit exécuté sur chaque processeur : en d'autres termes, le processus du code transformé «émule» (ou sérialise) l'exécution parallèle de plusieurs processus du code initial.

Nous avons validé cette transformation en l'appliquant au code VADOR et avec une grille test de quelques machines hétérogènes. Les résultats expérimentaux montrent qu'elle améliore sensiblement les performances du code initial sur la grille même en utilisant un modèle de performance très simple basé uniquement sur la vitesse des processeurs.

Cette transformation est décrite en deux étapes : nous donnons les conditions que doit vérifier le code MPI pour que la tranformation puisse s'appliquer, puis nous indiquons quelles sont les modifications syntaxiques à apporter au code.

3.4.1 Conditions d'application de la transformation

Notre transformation s'applique à un code SPMD qui respecte les conditions suivantes : *(i) le code peut s'exécuter avec un nombre quelconque de processeurs, (ii) la charge de travail est répartie équitablement entre les processeurs* et *(iii) la charge de travail dépend seulement du nombre de données à traiter.*

3.4.2 Modifications du code

Notre transformation est résumée par la figure 3.1. Elle consiste à modifier le code de façon à ce qu'un processus du code transformé émule plusieurs processus du code initial. Puisque le code initial est SPMD, il se décompose en une succession de *phases de calcul* et de *phases de communication*. Nous appelons ici phase de communication une séquence d'appels à des fonctions de communication MPI. Puisque le même code est exécuté par chaque processus, tout processus du code initial exécute une instance de ces phases successives. A gauche de la figure sont représentées les phases successives de deux processus du code initial : chacun des deux processus comporte quatre phases. A droite de la figure sont représentées les phases successives du processus qui émule les deux processus du code initial : les phases de calcul

sont regroupées ensemble et de même pour les phases de communication. Les appels à MPI

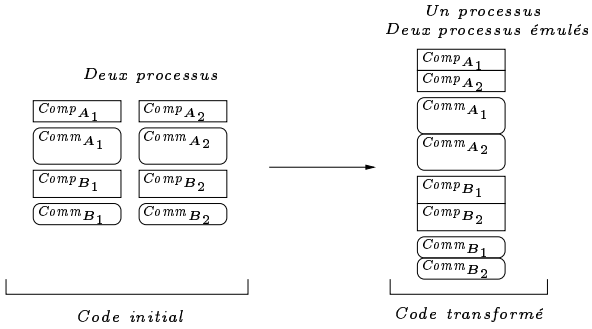


FIG. 3.1 – *Transformation de code.*

dans les phases de communication ont été modifiés de façon à remplacer les références aux processus émulés par les références aux processus réels et réaliser des copies mémoires plutôt que des communications lorsque la source et la destination sont le même processus du code transformé.

De plus, les communications point-à-point bloquantes (e.g. `MPI_Send`) ont été remplacées par leur version non-bloquante (e.g. `MPI_Isend`). Ceci a pour effet de laisser le système gérer les communications de façon à éviter les situations d'interblocage. Les doublons d'appel à une même communication collective doivent être supprimés : lorsqu'un processus du code initial fait appel à une communication collective, alors un processus du code transformé qui émule plusieurs processus ne doit faire qu'un seul appel à cette communication collective. Certaines communications collectives tel qu'un appel à `MPI_Barrier` ne requierent aucune autre modification, alors que d'autres nécessitent plus de modification du code. Par exemple, un appel à `MPI_Scatter` qui distribue à chacun des autres processus des morceaux de même taille d'un tableau de données, doit être remplacé par un appel à `MPI_Scatterv` qui permet une distribution inégale entre les processus du code transformé de façon à donner à chacun des autres processus du code transformé un morceau de taille correspondante au nombre de processus qu'il émule.

Exemple 13 Cet exemple expose plus en détail l'application de notre transformation de code. Nous considérons un morceau du code VADOR qui correspond au calcul d'un bloc de matrice suivi de sa transmission pour réaliser la transposition globale par bloc. Nous utilisons du pseudo-code pour décrire le contenu des processus.

Contenu d'un processus de rang r_i ($i = 1..2$) du code initial :

```
/* phase de calcul */
compute(A_i) ;

/* phase de communication */
MPI_Isend(..., A_i, dest_i, ...) ;
MPI_Irecv(..., B_i, src_i, ...) ;
```

Contenu d'un processus de rang r du code transformé :

```
/* phase de calcul */
for (i = 1..2)
    compute(A_i) ;

/* phase de communication */
for (i = 1..2) {
    if (proc_of(dest_i) ≠ r)  MPI_Isend(..., A_i, proc_of(dest_i), ...) ;
       else skip ;
    if (proc_of(src_i) ≠ r)  MPI_Irecv(..., B_i, proc_of(src_i), ...) ;
       else B_i = A_src_i ;
}
```

La phase de communication consiste à examiner la destination (resp. la source) des messages qui ont été envoyés (resp. reçus) par l'un des processus émulés. Si la source ou la destination est un processus émulé par le processus courant (r), alors une copie mémoire est réalisée (B_i = A_{src_i}). Si la source ou la destination est un processus émulé par un autre processus que r, disons r' (`proc_of(`$dest_i$`)` $\neq r$), alors une communication est réalisée entre r et r'. $\diamond$

Répartition de la charge : Le code transformé est paramétré par une *répartition de la charge* qui doit être connue au démarrage du code. Une répartition de la charge est définie par un p-uplet $(n_i)_{i\in[1..p]}$ d'entiers où n_i est le nombre de processus émulés qui doivent être alloués au processeur p_i, $i \in [1..p]$. Nous devons fournir au code transformé une répartition telle que le temps d'exécution total soit approximativement le même sur tous les processeurs.

3.5 Perspectives

Mes travaux futurs concerneront entre autres l'implantation des méthodes numériques adaptatives : les méthodes numériques adaptatives procèdent par discrétisation des équations sur un maillage de l'espace physique et sont capables de raffiner le maillage en fonction de critères mathématiques d'estimation de la solution. Ces méthodes permettent d'obtenir une efficacité optimale sur des machines séquentielles pour la plupart des familles d'EDP. Mais l'implantation effective de ces méthodes sur des architectures parallèles demeure un problème difficile. En effet, l'adaptivité de ces méthodes apporte de nouvelles complications : l'utilisation de maillages irréguliers et le fait que ces méthodes introduisent ou réduisent des degrés de liberté rendent l'équilibrage de charge plus difficile à réaliser.

L'implantation efficace de méthodes numériques complexes et adaptatives sur des architectures elles-mêmes complexes et adaptatives nécessite l'utilisation de concepts de programmation permettant de raisonner à différents niveaux d'abstraction et associés à des principes de structuration permettant de constituer des modules réutilisables et ainsi réduire la complexité grandissante lorsqu'il s'agit de simuler numériquement des systèmes physiques de plus en plus complexes.

Le data-parallélisme est un cadre naturellement approprié à la parallélisation des méthodes adaptatives puisque ces méthodes consistent essentiellement à appliquer un même traitement sur tous les éléments d'une structure de données globale.

Une partie de nos travaux pourra consisté en l'élaboration de transformations pour construire un programme parallèle en partant de problèmes exprimés à un haut niveau d'abstraction,

idéalement sous la forme d'un système d'équations différentielles enrichi de directives relatives au calcul par l'expertise du physicien ; Une autre partie à la généralisation des techniques de transformation aux cas irréguliers, architecture irrégulière (architectures évolutives de type grilles de calcul) (en interaction avec le projet TAG), ou structures de données irrégulières (structures creuses ou hiérarchiques).

Mes idées pour atteindre ces objectifs seront empruntées de mon cadre formel : la notion de *changement de base* pour exprimer la localité des données – l'expression de la localité des données constituant pour moi l'essence du parallélisme –, la géométrie support des transformations d'énoncés, la structuration basée sur la géométrie des objets, etc.

Plus précisément, l'idée directrice est que les techniques de parallélisation et en particulier celles associées aux méthodes adaptatives (notamment pour le partitionnement du maillage) reposent essentiellement sur des manipulations géométriques de domaines ou changements de base. La synthèse d'un algorithme parallèle consiste essentiellement en la définition de changements de base entre des domaines géométriques : le domaine des fonctions d'accès aux données du problème (le maillage dans le cas des méthodes adaptatives) et le domaine de l'architecture. Le raffinement du maillage définit lui aussi un changement de base. Une idée est donc de paramétrer tous les composants des méthodes adaptatives par un changement de base afin de ménager toutes les manipulations géométriques possibles. Cela peut être aussi une réponse complémentaire aux besoins de modularité et qui ne sacrifie pas à l'efficacité.

Bibliographie

[1] Projet TAG, Transformation et Adaptation pour la Grille, projet de l'ACI GRID `http ://grid.u-strasbg.fr`.

[2] Projet CALVI, CALcul scientifique et VIsualisation, avant projet INRIA `http ://www-irma.u-strasbg.fr/irma/general/simul/CALVI/calvi.html`.

[3] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution, 1993. In Workshop on Compilers for Parallel Computers, Delft, The Netherlands, December 1993.

[4] G. Antoniu, L. Bougé, R. Namyst, and C. Pérez. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. *Parallel Processing Letters*, 10(2/3) :201–207, 2000.

[5] R.J.R. Back. *On the correctness of refinement steps in program development.* PhD thesis, University of Helsinki, 1978.

[6] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(2) :593–624, 1988.

[7] U. Banerjee. *Loop Transformations for Restructuring Compilers : the Foundations.* Kluwer Academic Publishers, 1993.

[8] Siegfried Benkner and Thomas Brandes. Exploiting data locality on scalable shared memory machines with data parallel programs. *Lecture Notes in Computer Science*, 1900 :647–656, 2001.

[9] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of SuperComputing '96*, 1996.

[10] A. Bik and H. Wijshoff. Advanced compiler optimizations for sparse computations. *J. of Parallel and Distributed Computing*, 31 :14–24, 1995.

[11] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages : semantics and implementation. *FGCS*, 8 :363–378, 1992.

[12] K.M. Chandy and J. Misra. *Parallel Program Design :* A foundation. Addison Wesley, 1988.

[13] B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with features for locality control. Technical Report TR99-02, Inst. for Software Technology and Parallel Systems, U. Vienna www.par.univie.ac.at, February 1999.

[14] M. Chen and Y. Choo. Synthesis of a systolic Dirichlet product using non-linear domain contraction. Technical report, Yale University, December 1988.

[15] M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.

[16] C. Christara, X. Ding, and K. R. Jackson. An efficient transposition algorithm for distributed memory computers. In *13th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'99)*, pages 435–448, 1999.

[17] Ph. Clauss. *Synthèse d'algorithmes systoliques et implantation optimale en place sur réseaux de processeurs synchrones.* PhD thesis, Université de Franche-Comté, Mai 1990.

[18] Ph. Clauss, C. Mongenet, and G.-R. Perrin. Synthesis of size-optimal toroidal arrays for the algebraic path problem : A new contribution. *Parallel Computing*, 18 :185–194, 1992.

[19] L. Colombet. *Parallelization of applications for homogeneous and heterogeneous processors network.* PhD thesis, LMC-IMAG laboratory, Grenoble, October 1994. English version.

[20] O. Coulaud, E. Sonnendrücker, E. Dillon, P. Bertrand, and A. Ghizzo. Parallelisation of semi-lagrangian Vlasov codes. *J. Comput. Phys.*, 61, 1999.

[21] C. Creveuil. *Techniques d'analyse et de mise en oeuvre des programmes* GAMMA. PhD thesis, U. Rennes, 1991.

[22] A. Darte and Y. Robert. *Séquencement des nids de boucles.* Algorithmique parallèle. Masson, 1992.

[23] R. David, S. Genaud, A. Giersch, B. Schwarz, and E. Violard. Source code transformations strategies to load-balance grid applications. In *Third International Workshop on Grid Computing, GRID'2002*, volume LNCS 2536, pages 82–87. Springer-Verlag, June 2002.

[24] Jean-Luc Dekeyser and Philippe Marquet. Supporting irregular and dynamic computations in data parallel languages. In *LNCS Tutorial : The Data Parallel Programming Model*, volume 1132, pages 197–219. Springer-Verlag, 1996.

[25] P. Bertrand E. Sonnendrücker, J. Roche and A. Ghizzo. The semi-lagrangian method for the numerical resolution of Vlasov equations. *J. Comput. Phys.*, 149 :201–220, 1999.

[26] P. Bertrand F. Filbet, E. Sonnendrücker. Conservative numerical schemes for the Vlasov equation. J. Comput. Phys., 172 :166–187, 2001.

[27] P. Feautrier. Some efficient solutions to the affine scheduling problem, part 1, one-dimensional time. *Int. Journal of Parallel Programming*, 21(5) :313–348, 1992.

[28] *High Performance Fortran version 1.0*, January 1993.

[29] I. Foster and C. Kesselman. *The Grid, Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1998.

[30] Ian Foster and Nicholas Karonis. A grid-enabled MPI : Message passing in heterogeneous distributed computing systems. *Supercomputing*, November 1998.

[31] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.

[32] D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1) :80–112, January 1985.

[33] S. Genaud. *Transformations de programmes PEI : application au parallélisme de données.* PhD thesis, Université Louis Pasteur, Janvier 1997.

[34] Stéphane Genaud, Eric Violard, and Guy-René Perrin. Transformations techniques in PEI. *EUROPAR'95, LNCS*, 966 :131–142, August 1995.

[35] Ph. Gerner. *La sémantique des directives au compilateur : application au parallélisme de données.* PhD thesis, Université Louis Pasteur, Décembre 2002.

[36] Ph. Gerner and E. Violard. A theoretical framework of data parallelism and its operational semantics. In *EURO-PAR'2000*, volume LNCS 1900, pages 668–677. Springer-Verlag, September 2000.

[37] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The John Hopkins University Press, 1989. 2nd edition.

[38] P. Gribomont. Stepwise refinement and concurrency : the finite-state case. Technical report, PRLB, 1989.

[39] G. Hains, F. Loulergue, and J. Mullins. Concrete data structures and functional parallel programming. *Theoretical Computer Science*, 258(1–2) :233–267, April 2001.

[40] P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. *FPCA93, ACM Press*, pages 210–222, 1993.

[41] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25 :251–283, 1995.

[42] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of ACM*, 14(3) :563–590, July 1967.

[43] E. Knap. An exercise in the formal derivation of parallel programs : Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, Avril 1990.

[44] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*, January 1994.

[45] F. Kuijlman, H.J. Sips, C. van Reeuwijk, and W.J.A. Denissen. A unified compiler framework for work and data placement. In *Proc. of the ASCI 2002 Conference*, pages 109–115, Lochem, June 2002.

[46] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing : Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[47] C. Lengauer. Loop parallelization in the polytope model. *Parallel Processing Letters*, 4(3), 1994.

[48] V. Loechner. *Contribution à l'étude des polyèdres paramétrés et application en parallélisation automatique.* PhD thesis, U. Strasbourg I, 1997.

[49] F. Loulergue. *Conception de langages fonctionnels pour la programmation massivement parallèle.* PhD thesis, Université d'Orléans, France, Janvier 2000.

[50] C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.

[51] C. Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.

[52] G.-R. Perrin and E. Violard. Data parallelism and PEI equational language. Technical Report RR 99-01, LSIIT-ICPS, Université Louis Pasteur, http ://icps.u-strasbg.fr, May 1999.

[53] G.-R. Perrin, E. Violard, and S. Genaud. PEI : a theoretical framework for data parallel programming. In *Franco-British meeting on Data-Parallel Languages and Compilers*, April 1994.

[54] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989.

[55] S. Rajopadhye. LACS : A language for affine communication structures. Technical report, IRISA Rennes, 1993.

[56] The OpenMP Forum. OpenMP Fortran Application Program Interface. Proposal Ver 1.0, SGI, `http ://www.openmp.org`, October 1997.

[57] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. Technical report, Queen's University, October 1996. also published in Computing Surveys.

[58] G.D. Smith. Numerical solution of partial differential equations : finite difference methods. *Oxford Applied Mathematics and Computing series, Oxford University Press*, 1985.

[59] P. N. Swarztrauber. Transposing arrays on multicomputers using de Bruijn sequences. *J. Parallel Distributed Computing*, 53 :63–77, 1998.

[60] Thinking Machines Corp. *C⋆ Programming Guide*, November 1990.

[61] Thinking Machines Corp. *CM Fortran Programming Guide*, January 1991.

[62] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103, August 1990.

[63] E. Violard. *Une théorie unificatrice pour la construction de programmes parallèles par des techniques de transformations*. PhD thesis, Université de Franche-Comté, Octobre 1992.

[64] E. Violard. A mathematical theory and its environment for parallel programming, June 1993. Dagsthul Seminar 9325 on Parallelization Techniques for Uniform Algorithms organized by Ch. Lengauer, P. Quinton, Y. Robert, L. Thiele.

[65] E. Violard. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4 :313–328, 1994.

[66] E. Violard. Typechecking of PEI expressions. *EUROPAR'97, LNCS*, 1300 :521–529, 1997.

[67] E. Violard. What really is data parallelism ? Technical Report RR 00-01, LSIIT-ICPS, Université Louis Pasteur, January 2000.

[68] E. Violard, R. David, and B. Schwarz. Experiments in load balancing across the grid via a code transformation. In *DAPSYS 2002*, September 2002.

[69] E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694 :500–516, June 1993.

[70] E. Violard and G.-R. Perrin. PEI : a simple unifying model to design parallel programs. *Future Generation Computer Systems*, 10 :269–272, 1994. Elsevier.

[71] Eric Violard and Francis Filbet. Parallelization of a Vlasov solver by communication overlapping. In *Proceedings of PDPTA'2002*, pages 1049–1055. CSREA Press, June 2002.

[72] Eric Violard, Stéphane Genaud, and Guy-René Perrin. Refinement of data parallel programs in PEI. *IFIP TC2 Workshop on Algorithmic Languages and Calculi, Chapman & Hall*, February 1997.

[73] F. Voisin. *Etude d'outils logiciels pour la parallélisation et la transformation de programmes dans les applications de calculs numériques.* PhD thesis, Université Strasbourg I - Louis Pasteur, July 2001.

[74] F. Voisin and G.-R. Perrin. Sparse computations with PEI. *International Journal of Foundations of Computer Science*, 10(14) :425–442, 1999.

Annexe 1

Contexte ou collaborations

- Mes activités de recherche de nature théorique ont donné lieu à des échanges réguliers avec les meilleurs équipes françaises du domaine dans le cadre de l'opération inter PRC (C3 et ANM) ParaDigme, du GDR PRS (Parallélisme, Réseau et Systèmes) rebaptisé ARP (Architecture, Réseaux et systèmes, Parallélisme) et d'une ASP du MESR sur la programmation data-parallèle.

- Mes recherches sur l'utilisation des grilles de calcul s'inscrivent dans le cadre du projet TAG [1] mis en place dans l'équipe ICPS du laboratoire LSIIT. J'ai participé au lancement de ce projet. Stéphane Genaud est le coordinateur de ce projet qui est devenu officiellement un projet de l'ACI (Action Concertée Incitative du Ministère de la recherche) GRID (Globalisation des Ressources Informatiques et des Données). Ce projet est un projet pluridisciplinaire qui regroupe actuellement une équipe constituée de 2 ingénieurs, 2 doctorants, 2 maîtres de conférences, 2 professeurs et 1 chargé de recherches CNRS. Les laboratoires participants sont le LSIIT, l'IRMA, l'IPGS et l'ICS, tous ces laboratoires étant localisés à Strasbourg. Le projet TAG a pour objectif de proposer des techniques et outils permettant d'améliorer les performances des applications parallèles sur les grilles de calcul. Le projet repose sur l'idée de définir des transformations de code et de mesurer leur impact sur les performances. L'approche utilisée pour définir ces transformations est de considérer des applications réelles (dans différentes disciplines : en géophysique, physique des plasmas et chimie) et des codes MPI réels conçus pour des machines parallèles, et de valider ces transformations sur une grille test réaliste. J'ai proposé le code VADOR comme objet d'étude car il décrit un volume de calcul considérable et est un bon candidat pour la grille de calcul.

- Mes travaux sur l'implantation des méthodes numériques ont été effectués en collaboration avec Éric Sonnendrücker, Professeur à l'ULP, membre de l'IRMA (Institut de Recherche Mathématique Avancé, UMR 7501 ULP-CNRS) et Francis Filbet, chargé de recherche CNRS, tous deux mathématiciens appliqués. Ces travaux sont poursuivis dans le cadre du projet INRIA CALVI [2]. J'ai participé au lancement de ce projet dans lequel je suis fortement impliqué et dont le responsable est Éric Sonnendrücker. Ce projet

vise à faire interagir des mathématiciens appliqués et des informaticiens spécialistes de la visualisation et du calcul parallèle autour du thème de la simulation numérique de systèmes physiques complexes et qui évoluent au cours du temps (ex. l'évolution d'un plasma dans un tokamak). Ces systèmes sont modélisés par des équations à dérivées partielles. Les systèmes physiques considérés ont en commun la très grande quantité de calculs nécessaire à leur simulation compte tenu : de la présence d'échelles multiples, du très grand nombre de points du maillage qui sont à traiter afin d'obtenir une précision acceptable, et du grand nombre de dimensions de l'espace considéré pour des résultats réalistes. Il est bien sûr nécessaire de recourir au parallélisme et à des environnements d'exécution de grande échelle pour obtenir la meilleure efficacité et faire face à l'enrichissement des modèles pour inclure des effets toujours plus réalistes. Nous envisageons l'utilisation de grilles de calcul pour répondre aux besoins en puissance de calcul.

Annexe 2

Activités d'encadrement

À ce jour, mes activités d'encadrement consistent en :

- l'encadrement de 4 stages de TER (Travail d'Etude et de Recherche) en Maîtrise :
  - 3, en 2000-2001 dont 1 sur un thème autour de la théorie PEI :
    «résolution de contraintes de placement de données
    sur une grille de processeurs virtuels».
  - 1, cette année, sur le thème
    «modèle de performance des versions parallèles du code VADOR».

- l'encadrement de 2 stages de DESS en collaboration avec le CEA en 1994-1995 sur le thème programmation parallèle de problèmes industriels en utilisant PEI. Ce travail a montré l'intérêt pratique de PEI pour développer du code pour matrices creuses, il a été poursuivi et a mené à 2 publications dont 1 en revue internationale.

- l'encadrement de 4 stages de DEA :
  - 1, en 1995-1996, sur le thème
    «étude comparative de PEI et CRYSTAL».
  - 1 (stagiaire Philippe Gerner), en 1996-1997, sur le thème
    «sémantique opérationnelle des énoncés en PEI».
  - 1 (stagiaire Arnaud Giersch), en 1999-2000, sur le thème
    «preuve de programmes data-parallèles».
  - 1, en 2001-2002, sur le thème
    «automatisation des règles de transformations en BMF (Bird-Merteens Formalism)».

- le co-encadrement de la thèse de **Stéphane Genaud** soutenue le 9 janvier 1997 sur le thème «transformations de programmes PEI et application au parallélisme de données».

- le co-encadrement au début de la thèse d'Arnaud Giersch (actuellement en 3ème année de thèse) sur le thème de l'adaptation de codes MPI à la grille de calcul. Ces recherches s'inscrivent dans le cadre du projet TAG.

- l'encadrement de la thèse de **Philippe Gerner** soutenue le 20 décembre 2002 sur le thème spécification de la coopération entre programmeur et compilateur. Ph. Gerner a étudié les rapports entre la sémantique opérationnelle «officielle» des programmes et leur exécution une fois compilés. Dans sa thèse, il considère en particulier le cas de langages parallèles à directives comme HPF où les directives ne sont pas nécessairement respectées par le compilateur. Il propose un cadre pour exprimer et étudier le contrat nécessaire entre programmeur et compilateur. Ce cadre est appliqué à l'étude des langages data-parallèles. Une partie de ces travaux concernant la sémantique opérationnelle des langages data-parallèles ont donné lieu à une publication en conférence internationale.

  Philippe Gerner poursuit ses travaux de recherche dans le cadre d'un post-doctorat au laboratoire VERIMAG à Grenoble sous la direction de J. Sifakis toujours sur le thème de la prise en compte et la représentation de contraintes d'exécution, mais dans le cadre des systèmes embarqués temps réel encore plus riche en contraintes.

Annexe 3

Liste des publications d'Éric Violard

Revues internationales

E. Violard et G.-R. Perrin. «PEI : a language and its refinement calculus for parallel programming» *Parallel Computing*, Vol.18, 1167–1184 (1992)

E. Violard. «A mathematical theory and its environment for parallel programming» *Parallel Processing Letters*, Vol.4(3), 313–328 (1994)

E. Violard et G.-R. Perrin. «PEI : a simple unifying model to design parallel programs» *Future Generation Computer Systems*, 10 (1994) 269–272, Elsevier.

E. Violard. «A Semantic Framework To Adress Data Locality in Data Parallel Languages» à paraître dans *Parallel Computing* (2003).

Conférences internationales **(avec comité de lecture et actes publiés)**

E. Violard et G.-R. Perrin. «PEI : a single unifying model to design parallel programs» *Parallel Architecture and Languages Europe*, PARLE'93, Munich, LNCS 694 (1993), 500–516, Springler-Verlag.

E. Violard. «Reduction in PEI». *Third Joint International Conference on Vector and Parallel Processing*, CONPAR'94 VAPP VI, Linz, LNCS 854 (1994), 112–123, Springler-Verlag.

S. Genaud, E. Violard et G.-R. Perrin. «Transformations techniques in PEI», *First International EURO-PAR Conference*, EUROPAR'95, Stockholm, LNCS 966 (1995), 131–142, Springer-Verlag.

E. Violard, S. Genaud et G.-R. Perrin. «Refinement of data parallel programms in PEI», *Working Conference on Algorithmic Language and Calculi*, IFIP-WG 2.1, Le Bischenberg (France),

In proceedings of the 50th meeting (1997), 25 pages, Chapman & Hall Eds.

E. Violard. «Typechecking of PEI expressions», *Third International EURO-PAR Conference*, EUROPAR'97, Passau, LNCS 1300 (1997), 521–529, Springer-Verlag.

Ph. Gerner et E. Violard. «A Theoretical Framework of Data Parallelism and its Operational Semantics», *6th International EURO-PAR Conference*, EUROPAR'2000, Munich, LNCS 1900 (2000), 668–677, Springer-Verlag.

E. Violard et F. Filbet. «Parallelization of a Vlasov Solver by Communication Overlapping» *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'2002, Las Vegas, 1049–1055, CSREA Press.

E. Violard, R. David et B. Schwarz. «Experiments in Load Balancing Across the Grid via a Code Transformation». *Fourth Austrian-Hungarian Workshop on Distributed and Parallel Systems*, DAPSYS'2002, Linz, 66–73, Kluwer.

R. David, S. Genaud, A. Giersch, B. Schwarz et E. Violard. «Source Code Transformations Strategies to Load-balance Grid Applications», *Third International Workshop on Grid Computing*, GRID'2002, Baltimore, LNCS 2536 (2002), 82–87, Springer-Verlag.

Séminaires internationaux sur invitation

Dagstuhl Seminar 9325 on Parallelization Techniques for Uniform Algorithms (Juin 1993)

Dagstuhl Seminar 9708 on Theory and Practice of Higher-Order Parallel Programming (Février 1997)

Communications

E. Violard. «PEI : un modèle unificateur simple pour la conception de programmes parallèles» Communication à RenPar'5, Brest (Mai 1993)

G.-R. Perrin, E. Violard et S. Genaud. «PEI : a theoretical framework for data-parallel programming» Communication au Workshop Franco-Britanique «Data-parallel Languages and Compilers", Lille (Avril 1994)

Rédaction de projets de recherche

Participation à la rédaction du projet TAG [1] (actuellement projet ACI GRID).
Participation à la rédaction du projet CALVI [2] (actuellement projet INRIA).

Présentations, posters, séminaires

Présentation de l'environnement PEI à Sophia-Antipolis, Nice (1993)

Forum des Recherches en Informatique - Ecole Polytechnique, Paris (02-03 Juin 1993)

Poster à HPCN, Amstersdam (1993)

Présentation du projet TAG à EDF Recherche & Développement à Clamart
dans le cadre du séminaire «Calcul Scientifique Distribué et Grid computing»
(20 mars 2002).

Séminaire interne LSIIT-ICPS «Parallélisation d'un code en Physique des Plasmas»
(8 mars 2002).

Rapports de recherche

E. Violard. «Data-parallelism vs Functional Programming : the contribution of PEI» (1995)

E. Violard. «Asynchronous Parallel Programming in PEI» (1997)

E. Violard. «A formal semantics of data parallel languages» (1998)

G.-R. Perrin et E. Violard. «Data Parallelism and PEI Equational Language» (1999)

E. Violard. «What Really is Data Parallel Programming ?» (2000)

Réalisations

Environnement PEI pour la conception de programmes parallèles mis en oeuvre en CENTAUR (CENTAUR est un produit INRIA).

Un contrôleur de type pour le langage PEI, mis en oeuvre en CAML en utilisant la librarie OMEGA (de William Pugh).

Développement de plusieurs versions du code VADOR en C++/MPI pour machines parallèles et pour grilles de calcul.

Annexe 4

Publications sur PEI
(par d'autres auteurs)

Revue internationale

F. Voisin et G.-R. Perrin. «Sparse Computation with PEI», International Journal of Foundations of Computer Science, Vol 10(4), 425–444 (1999).

Revues nationales

S. Genaud. «Transformations d'énoncés PEI», Technique et Science Informatiques, Vol. 15(5), 601-618, Hermes (1996).

Thèses ou chapitres de thèse

S. Genaud. «Transformations de programmes PEI : applications au parallélisme de données», Université Louis Pasteur (Janvier 1997).

F. Voisin. «Étude d'outils logiciels pour la parallélisation et la transformation de programmes dans les applications de calculs numériques», Université Louis Pasteur (Juillet 2001).

Ph. Gerner. «La sémantique des directives au compilateur : application au parallélisme de données», Université Louis Pasteur (Décembre 2002).

Communications

S. Genaud. «Techniques de transformations d'énoncés PEI pour la production de programmes data-parallèles», Communication à RenPar'7, Mons (Belgique) (Mai 1995).

F. Voisin. «Minimisation des communications dans une résolution distribuée des équations de Navier-Stokes», Comunication à Renpar'10, Strasbourg (Juin 1998).

F. Voisin et G.-R. Perrin. «Sparse Computation with PEI» , 6th International Workshop on Solving Irregularly Structured Problems in Parallel, San Juan (Puerto Rico), In proceedings of IPPS/SPDP'99 Workshops, LNCS 1586 (1999), Springer-Verlag.

Réalisations

Visual PEI - Un outil de conception graphique d'énoncés PEI
réalisé par Stéphane Genaud.

Un translateur PEI - HPF écrit en CAML
réalisé par Stéphane Genaud.

Un interpréteur PEI écrit en CAML
réalisé par Philippe Gerner.

Annexe 5

Curriculum-Vitae

Éric Violard
né le 23 décembre 1966 à Montbéliard (Doubs)
Vit maritalement, 1 enfant

Formation :

1984 Baccalauréat série C à Montbéliard (25).
1984 DEUG option Maths, Université de Franche-Comté, Besançon.
1987 Licence Informatique, mention bien, Université de Franche-Comté, Besançon.
1988 Maîtrise d'informatique mention bien, Université de Franche-Comté, Besançon.
1988 DEA Automatique, Informatique et Robotique mention bien, Université de Franche-Comté, Besançon.
1992 Thèse de Doctorat en informatique mention très honorable soutenue le 26 octobre 1992, Université de Franche-Comté, Besançon. Titre : *Une théorie unificatrice pour la construction de programmes parallèle par des techniques de transformations.* Directeur de thèse : Guy-René Perrin. Jury : Patrice Quinton (président), Pascal Gribomont (rapporteur), Daniel Le Métayer (rapporteur), Didier Arquès (examinateur), Guy-René Perrin (examinateur).

Prises de fonction :

- Maître de conférences à l'Université de Franche-Comté (Besançon) de 1993 à 1995.
- Maître de conférences à l'Université Louis Pasteur (Strasbourg) depuis 1995.

Responsabilités collectives :

- Responsable de la filière IUP 1ère année depuis 2001.
- Membre titulaire de la CS section 27 de l'ULP depuis 2002. Membre suppléant en 2001.
- Membre titulaire de la CS section 27 de l'École des Mines de Nancy (INPL) depuis 2000.

Activités d'enseignement :

- Cours de programmation parallèle en DESS informatique.
- Cours de dérivation de programmes parallèles en DEA informatique.

- Cours de spécifications formelles et preuves de programmes en Maîtrise d'informatique.
- Cours, TD et TP d'algorithmique et programmation impérative en IUP GMI 1ère année.
- TD de programmation fonctionnelle en Licence et Maîtrise d'informatique.

Annexe 6

Publications jointes

Dans cette annexe se trouvent quelques publications résumant mes travaux :

E. Violard. «A Semantic Framework to Adress Data Locality in Data Parallel Languages» à paraître dans *Parallel Computing* (2003).

E. Violard. «A mathematical theory and its environment for parallel programming» *Parallel Processing Letters*, Vol.4(3), 313–328 (1994)

E. Violard, S. Genaud et G.-R. Perrin. «Refinement of data parallel programms in PEI», *Working Conference on Algorithmic Languages and Calculi*, IFIP-WG 2.1, Le Bischenberg (France), In proceedings of the 50th meeting (1997), 25 pages, Chapman & Hall Eds.

E. Violard et F. Filbet. «Parallelization of a Vlasov Solver by Communication Overlapping» *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'2002, Las Vegas, 1049–1055, CSREA Press.

E. Violard, R. David et B. Schwarz. «Experiments in Load Balancing Across the Grid via a Code Transformation». *Fourth Austrian-Hungarian Workshop on Distributed and Parallel Systems*, DAPSYS'2002, Linz, 66–73, Kluwer.

Ph. Gerner et E. Violard. «A Theoretical Framework of Data Parallelism and its Operational Semantics», *6th International EURO-PAR Conference*, EUROPAR'2000, Munich, LNCS 1900 (2000), 668–677, Springer-Verlag.