Transformations d'énoncés PEI

Stéphane Genaud

ICPS - Université Louis Pasteur, Strasbourg
Pôle API
Bd Sébastien Brant
67400 Illkirch

RÉSUMÉ. *Nous présentons le formalisme* PEI *et nous montrons qu'il représente un outil adéquat pour la spécification et la transformation de programmes data-parallèles. Le formalisme repose sur un modèle géométrique et des transformations basées sur un calcul de raffinement permettent de mettre en évidence l'équivalence de programmes. Les concepts de* PEI *permettant d'exprimer les fondements de la programmation data-parallèle, nous montrons comment les énoncés peuvent être transformés et traduits dans un langage data-parallèle.*

ABSTRACT. *We present the* PEI *formalism and we show its adequacy to data parallel programs specification and transformation. The formalism lies on a geometrical model and transformations based on a refinement calculus are used to put forward programs equivalence. As* PEI *concepts include the data parallel paradigm, we show how* PEI *statements can be transformed and translated into a target data parallel language.*

MOTS-CLÉS : *Transformation de programmes, raffinement, abstraction, diffusion, réduction, data-parallélisme.*

KEY WORDS : *Program transformation, refinement, abstraction, broadcast, reduction, data parallelism.*

1. Introduction

Le modèle de programmation à parallélisme de données [TSI 93], ou *data-parallèle*, présente des qualités intrinsèques pour la programmation de certaines applications scientifiques : la manipulation de structures de données et de contrôle régulières, dans un modèle d'exécution essentiellement synchrone, permet en effet une réécriture relativement aisée des programmes séquentiels. C'est pourquoi des extensions des langages classiques au *data-parallélisme* sont proposées (cf. HPF [HIG 93], par exemple), sous la forme de directives d'alignement, de distribution des *variables parallèles* sur des grilles de *processeurs virtuels* et d'opérations globales sur ces variables qui expriment, généralement implicitement, les communications nécessaires. Une telle approche est certainement porteuse d'espoir d'une normalisation de ces langages et, par conséquent, de progrès significatifs en matière de développement ou de portage de codes.

Cependant, derrière cette apparente facilité, se cachent des questions délicates pour la maîtrise du modèle de programmation, pour la construction, la transformation ou la preuve des programmes data-parallèles, et leur mise en œuvre efficace sur des architectures diverses. Ces études requièrent une définition des fondements du modèle même. A côté d'autres travaux sur la sémantique de ces langages ou la preuve des programmes (cf. [BOU 94], par exemple), c'est une telle définition que propose le formalisme PEI. Conçu dès l'origine [VIO 92, VIO 94B] comme un cadre formel de raffinement des programmes qui généralise les langages d'équations récurrentes comme ALPHA [MAU 89] ou CRYSTAL [CHE 91], PEI se trouve être un cadre adéquat pour la conception et la transformation de programmes data-parallèles. Le lien entre énoncés PEI et leur équivalent dans un langage data-parallèle est quasi immédiat : nous le montrerons tout au long de l'article par des traductions en HPF.

Grâce au suppport formel de PEI, les énoncés peuvent être transformés sur la base d'un calcul de raffinement, permettant ainsi de mettre en évidence plusieurs programmes satisfaisant un même problème. Ce calcul de raffinement est défini à partir de règles élémentaires de transformations : dans la pratique, le processus de transformation de programmes requiert de combiner un certain nombre de ces règles, jusqu'à obtenir une version jugée satisfaisante par le programmeur. C'est ainsi que dans cet article, nous nous attachons plus particulièrement à la transformation des fragments de programmes traitant des communications, de façon à faire apparaître des solutions adaptées au modèle de programmation data-parallèle. Parmi celles qu'il est intéressant d'introduire, on peut naturellement penser aux solutions utilisant des opérations *scan* ou des *réductions*, dont la mise en œuvre, sur une machine à p processeurs, permet de résoudre des problèmes de taille n en un temps de l'ordre de $\frac{n}{p} + \log_2(p)$ [KRU 85].

La manière dont on combine les règles de transformations constitue ce qu'on peut appeler des *stratégies de transformation*. L'objectif de cet article est de présenter quelques-unes de ces stratégies, en précisant les hypothèses de leur application. Pour cela, après un bref rappel du formalisme PEI, l'article décrit le calcul de raffinement régissant les transformations. Suit un exemple de transformations de programme pour illustrer deux stratégies très générales. Le programme considéré calcule le produit matrice-vecteur : une première version utilise des communications explicitées par le programmeur, tandis que la version finale, obtenue après deux transformations successives, n'utilise plus que des opérations de diffusion et de réduction.

2. Définition du formalisme PEI

Depuis quelques années, un certain nombre de travaux sur l'analyse statique des programmes ont abouti à des techniques de parallélisation des programmes : recherche de boucles parallèles, distribution des calculs et des données, *etc.* Les techniques de réécriture de nids de boucles, en vue de leur parallélisation, reposent sur des manipulations de nature géométrique du domaine d'itération ou des domaines d'indices des tableaux, sous des hypothèses fortes, comme la linéarité des fonctions de dépendance et des bornes d'itération, par exemple. Des considérations du même type se retrouvent en programmation data-parallèle, dans les études sur l'alignement et la distribution des données, ou des communications induites. Ces remarques montrent l'importance de la géométrie dans ces approches et l'intérêt de pouvoir manipuler les indices des tableaux de manière abstraite, au même titre que les définitions des calculs, en exprimant le placement des valeurs dans un *référentiel* qui permette de distinguer chacune d'elles par des coordonnées discrètes. La transformation du placement des valeurs peut alors permettre de définir des calculs efficaces sur une architecture parallèle donnée.

C'est ainsi que l'activité de programmation peut fondamentalement reposer sur un très petit nombre de primitives nécessaires à l'écriture des programmes. Nous en mettons trois en évidence :

— l'application de fonctions sur des valeurs, pour définir les calculs,

— le déplacement de valeurs dans le référentiel, pour exprimer les dépendances, et par conséquent les éventuelles communications,

— le changement de référentiel, pour transformer la structure des données et des calculs.

Ces idées très simples sont le fondement du formalisme PEI que nous rappelons maintenant.

2.1. *Introduction à la programmation* PEI

De manière générale, considérons qu'un problème est spécifié par une relation entre des *multi-ensembles* de valeurs, appelés ses *entrées* et ses *sorties*. En PEI, les valeurs d'un multi-ensemble sont placées sur un domaine discret pour former un *champ de données*[1]. Par exemple, le multi-ensemble de valeurs $A = \{1, -2, 3, 1\}$ peut être exprimé par le champ de données `VECTA`, dont chaque élément est distingué par un indice de $\mathbb{Z}$ (entre 0 et 3 par exemple). On peut définir ou transformer une expression en appliquant des *opérations*

1. A la façon des *tableaux* dans les langages impératifs

qui changent soit les valeurs, soit le placement de ces valeurs sur le domaine discret, soit le domaine discret lui-même.

Pour cela, on écrit des équations non orientées, chacune définissant l'égalité d'expressions de champs de données. Ainsi, l'équation `VECTB = VECTA` indique que le champ `VECTB` a les mêmes valeurs que `VECTA`, et que ces valeurs sont placées de la même manière sur le même domaine discret. L'ensemble des équations constitue un *programme* s'il existe au plus une solution, c'est-à-dire que pour un jeu de données en entrée, il y a au plus un jeu de multi-ensembles de valeurs en sortie. Par la suite, nous noterons $P\{S\}$ un programme P ayant S pour ensemble d'équations.

Nous pouvons d'ores et déjà illustrer à l'aide de quelques exemples simples, les primitives du langage :

Exemple 1 *Addition d'une matrice et de sa transposée*

$$\mathtt{MATSUM : A} \mapsto \mathtt{C}$$

$$\left\{\begin{array}{l} \mathtt{A} = \mathtt{matrix :: A} \\ \mathtt{T} = \mathtt{A} \lhd \mathtt{transp} \\ \mathtt{C} = \mathtt{add} \rhd \mathtt{(A\ /\&/\ T)} \end{array}\right.$$

$$\begin{array}{lcl} \mathtt{matrix} & = & \lambda\mathtt{(i,j)|(0<=i,j<n).(i,j)} \\ \mathtt{transp} & = & \lambda\mathtt{(i,j)|(0<=i,j<n).(j,i)} \\ \mathtt{add} & = & \lambda\mathtt{(a{\cdot}b).(a+b)} \end{array}$$

— la première équation est une précondition sur le champ de données en entrée `A`. Elle exprime que `A` n'est pas modifié par l'application du *changement de base* `matrix` (l'application de ce changement de base est dénotée par le symbole `::`). Ce changement de base est l'identité sur le domaine carré $(0 \leq i < n, 0 \leq j < n)$: comme nous le verrons par la suite, ceci signifie que le domaine discret sur lequel on place les valeurs de `A` est inclus dans ce carré.

— la deuxième équation définit le champ `T` par une *opération de routage* appliquée sur `A` (l'application de cette opération est dénotée par le symbole $\lhd$). De manière générale, un routage exprime des dépendances sur le domaine discret : le champ $\mathtt{X} \lhd \mathtt{g}$ est tel que ses valeurs de coordonnées $z \in dom(\mathtt{g})$ sont issues des points de coordonnées $\mathtt{g}(z)$ de `X`. En l'occurrence sur l'exemple, les valeurs placées aux points de coordonnées (i, j) dans `T` sont celles placées en (j, i) dans le champ `A`.

— la troisième équation définit le champ de données en sortie `C`. La partie droite définit la *superposition* de `A` et `T` (notation `/&/`), *i.e.* l'ensemble des couples $\mathtt{(a{\cdot}b)}$, avec $\mathtt{a} \in \mathtt{A}$ et $\mathtt{b} \in \mathtt{T}$. L'*opération fonctionnelle* `add` (dont l'application est dénotée par le symbole $\rhd$) est appliquée sur tous les éléments de ce champ.
□

Exemple 2 *Définition récursive d'un champ de données*

Voyons maintenant un exemple un peu plus complexe, comportant une définition récursive de champ de données. Ce type de définition permet par exemple de définir des opérations de type *parallèle préfixe* comme le scan ou la réduction, quand la récurrence utilise un opérateur binaire associatif. Ainsi, le programme suivant calcule les sommes partielles d'un ensemble d'entiers [2] :

$$\texttt{PREFIX-SUM : A} \mapsto \texttt{R}$$
$$\begin{cases} \texttt{A} = \texttt{vector :: A} \\ \texttt{R} = \texttt{add} \rhd \texttt{(A /\&/R} \lhd \texttt{prev)} \end{cases}$$
$$\begin{array}{lcl} \texttt{vector} & = & \lambda\texttt{(i)}|\texttt{(0<=i<n)} \\ \texttt{prev} & = & \lambda\texttt{(i)}|\texttt{(0<i<n).(i-1)} \\ \texttt{add} & = & \texttt{id} \# \lambda\texttt{(a}\cdot\texttt{b).(a+b)} \end{array}$$

La figure 1 montre que le champ `R` contenant les sommes partielles des entiers consécutifs de 1 à 6 est solution de l'équation $\texttt{R} = \texttt{add} \rhd \texttt{(A /\&/R} \lhd \texttt{prev)}$ pour le champ `A` donné sur l'exemple. Dans le cas où on cherche seulement à

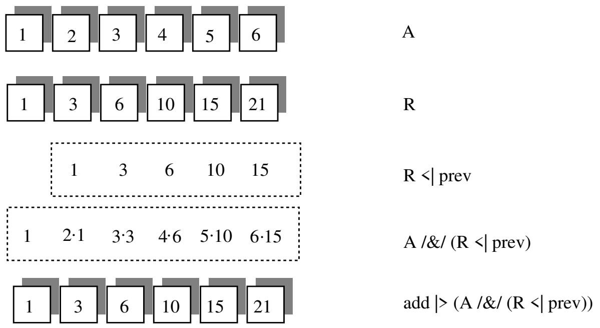

Figure 1. *Le champ* `R` *contient les sommes partielles du champ* `A`

calculer la somme des n éléments, on introduit un champ de données `T` dont la définition est $\texttt{T} = \texttt{R} \lhd \texttt{last}$, avec $\texttt{last} = \lambda\texttt{(i)}|\texttt{(i=n-1)}$. Un tel champ peut également être défini par le programme suivant :

$$\texttt{RED-SUM : A} \mapsto \texttt{T}$$
$$\begin{cases} \texttt{A} = \texttt{vector :: A} \\ \texttt{T} = \texttt{sum} \rhd \texttt{(red} \cdot \rhd \texttt{A)} \end{cases}$$

2. La fonction prédéfinie `id`, utilisée dans ce programme, est une fonction qui, appliquée à une séquence formée d'un seul élément, rend cet élément. Le symbole # indique que la fonction est définie par parties sur des domaines disjoints : `sum`, sur le domaine des séquences à un élément, rend cet élément. Sur le domaine des séquences à deux éléments (λ(`a·b`)), elle rend `(a+b)`.

```
vector  =  λ(i)|(0<=i<n)
red     =  λ(i)|(0<=i<n).(n-1)
sum     =  id # λ(a·b).(a+sum(b))
```
□

Ce programme montre un exemple de *réduction géométrique* en PEI. De manière générale, la réduction géométrique (dont l'application est dénotée $\mathtt{g} \cdot \triangleright \mathtt{X}$) est définie comme l'inverse de l'opération de routage, quelle que soit la fonction : elle place en tout point de coordonnées z, une séquence des valeurs de $\mathtt{X}$ situées en tous les points y, $y \in dom(\mathtt{g})$, tels que $\mathtt{g}(y) = z$. Sur l'exemple, la fonction `red` appliquée à l'ensemble des valeurs placées dans l'intervalle $[0, .., n-1]$ rend la séquence des valeurs (dans un ordre indéterminé) et la place au point $i = n - 1$.
Introduisons tout de suite une propriété importante de cette opération de réduction, qui sera utilisée pour montrer la relation entre des programmes utilisant les opérations de routage et de réduction. Cette propriété rappelle qu'une réduction qui n'implique qu'un point est exactement un routage par la fonction inverse.

Proposition 1. *Le champ de données* $\mathtt{g} \cdot \triangleright \mathtt{X}$ *est égal à* $\mathtt{X} \triangleleft \mathtt{g}^{-1}$ *si et seulement si* `g` *est bijective.*
◇

2.2. *Objets du langage*

Nous venons de voir intuitivement quelques programmes PEI. Ils suffisent à comprendre que les seuls objets manipulés en PEI sont les champs de données et les fonctions qui permettent d'écrire les opérations appliquées aux champs de données. La notation des fonctions en PEI est dérivée du lambda-calcul. Ainsi une fonction `f` de domaine $dom(\mathtt{f}) = \{\mathtt{x} \mid \mathtt{P(x)}\}$ est notée par l'expression $\lambda\mathtt{x}|\mathtt{(P(x)).f(x)}$. Par commodité, nous écrivons $\lambda\mathtt{x.f(x)}$ pour $\lambda\mathtt{x}|\mathtt{(true).f(x)}$, et $\lambda\mathtt{x}|\mathtt{(P(x))}$ pour $\lambda\mathtt{x}|\mathtt{(P(x)).x}$. Enfin, une fonction `f` définie sur des sous-domaines disjoints est notée `f1 # f2`, et le domaine d'une fonction composée $\mathtt{f} \circ \mathtt{g}$ est $\{\mathtt{x} \in dom(\mathtt{g}) \mid \mathtt{g(x)} \in dom(\mathtt{f})\}$.

2.3. *Sémantique*

Un champ de données est un placement des valeurs d'un multi-ensemble de valeurs V sur un ensemble discret, tel que l'on puisse distinguer chacune des valeurs par des coordonnées entières. Ce placement est une application $v : \mathbb{Z}^n \mapsto V$. Nous avons parlé du champ de données `VECTA` dont les valeurs $\{1, -2, 3, 1\}$ sont placées dans cet ordre sur l'intervalle $[0..3]$ de $\mathbb{Z}$. Le même multi-ensemble de valeurs peut être exprimé par un autre champ de données, par exemple **MATA** qui placerait les valeurs sur les points de coordonnées $0 \leq i, j < 2$ de $\mathbb{Z}^2$. Ces deux champs de données diffèrent par le placement des valeurs, et il y a évidemment une bijection entre ces deux placements, par exemple: $\sigma(i) = (i \ mod \ 2, i \ div \ 2)$. Si $v : \mathbb{Z} \mapsto V$ est le placement de **VECTA** et $v' : \mathbb{Z}^2 \mapsto V$ celui de **MATA**, on a $v = v' \circ \sigma$. C'est pourquoi nous caractérisons un champ de données `X` par une paire de fonctions $(v : \sigma)$ où σ est une bijection de $\mathbb{Z}^n$ dans $\mathbb{Z}^p$, telle que $dom(v) \subseteq dom(\sigma)$. Ceci permet de définir l'*interprétation fonctionnelle* $[\![\mathtt{X}]\!]$ d'un champ de données `X` comme étant $v \circ \sigma^{-1}$. Cette interprétation définit les valeurs d'un champ de données quel que soit le placement des valeurs.

Outre la *superposition*, opérateur interne associatif noté `/&/` qui forme des séquences à partir des valeurs de ses arguments, la sémantique des trois opérations externes sur un champ $\mathtt{X}= (v : \sigma)$ est définie par :

— soit `f` une fonction de V dans un autre ensemble de valeurs W, l'opération fonctionnelle $\mathtt{Y = f} \rhd \mathtt{X}$ définit `Y` comme $(\mathtt{f} \circ v : \sigma)$;

— soit `g` une fonction partielle de $\mathbb{Z}^n \mapsto dom(v)$, alors l'opération de routage $\mathtt{Y = X} \lhd \mathtt{g}$ définit `Y` comme étant $(v \circ \mathtt{g} : \sigma)$;

— soit `h` une bijection de $\mathbb{Z}^n \mapsto \mathbb{Z}^p$ définie sur $dom(v)$, le changement de base `Y = h :: X` définit `Y` comme étant $(v \circ \mathtt{h}^{-1}\!: \sigma \circ \mathtt{h}^{-1})$.

Le changement de base permet d'introduire l'équivalence de champs de données : à condition que $dom(\sigma) \subseteq dom(\mathtt{h})$ le champ de données $\mathtt{X}=(v : \sigma)$ est équivalent au champ `h::X` puisque ces deux placements sont ceux d'un même multi-ensemble de valeurs. On vérifie évidemment que deux champs équivalents ont la même interprétation fonctionnelle, c'est-à-dire $[\![\mathtt{h :: X}]\!] = [\![\mathtt{X}]\!] = v \circ \sigma^{-1}$.

3. Transformation de programmes

3.1. *Raffinement et abstraction*

Le raffinement est un procédé classique de construction ou de transformation de programmes [BAC 88, MOR 90], qui consiste à obtenir un programme satisfaisant toute spécification que satisfaisait un programme initial. Cette notion est généralement étendue aux spécifications elles-mêmes, dans un procédé continu de construction d'énoncés successifs depuis une spécification jusqu'aux programmes. Ceci définit une relation d'ordre partiel entre énoncés, dite *relation de raffinement*, dont l'inverse s'appelle *l'abstraction*, et caractérisée par la propriété de *satisfaction* choisie (par exemple la correction partielle, en programmation impérative).

De manière analogue nous définissons une relation de raffinement entre programmes PEI : le programme $P\{S\}$ est dit raffiné par le programme $P\{S'\}$, et on note $P\{S\} \sqsubseteq P\{S'\}$, si toute solution de S' est une solution de S. De plus, si $P\{S\}$ et $P\{S'\}$ se raffinent mutuellement, nous dirons que ces programmes sont équivalents, ce que nous notons $P\{S\} \equiv P\{S'\}$.

A titre d'exemple, on peut observer intuitivement que le programme **RED-SUM** présenté en section 2.1 est raffiné par le programme **PREFIX-SUM** (auquel il convient d'ajouter la définition de `T`). En effet le déplacement, de proche en proche, des valeurs par le routage **prev** est une façon particulière d'opérer une réduction au point de coordonnée $i = n - 1$. Pour un même champ de données en entrée **A** de ces deux programmes, le champ de données en sortie **T** de **PREFIX-SUM** est donc également solution du programme **RED-SUM**.

Cette observation, simple sur l'exemple considéré, doit en fait être conduite rigoureusement si on veut en faire un procédé général de transformation de programmes (soit pour les raffiner, soit pour les abstraire). Pour cela, on doit définir -et prouver- des règles élémentaires de transformation, et montrer qu'on peut les combiner pour définir des transformations plus globales. C'est ainsi qu'en constatant que la fonction **red**, introduite dans le programme **RED-SUM**, peut se décomposer récursivement (sans introduire pour l'instant de condition d'arrêt) en **red** $\circ$ **succ**, où **succ=**λ**(i)|(0<=i<n-1).(i+1)**, on est amené à chercher à réécrire l'expression **(red** $\circ$ **succ)** $\cdot\triangleright$ **A** sous une autre forme. Une règle élémentaire va permettre cette réécriture, qui précise que l'on peut substituer une équation de la forme `Y = (g1` $\circ$ `g2)` $\cdot\triangleright$ `X` par l'équation `Y = g1` $\cdot\triangleright$ `(g2` $\cdot\triangleright$ `X)`. On est alors ramené à la définition du champ de données **red** $\cdot\triangleright$ **(succ** $\cdot\triangleright$ **A)**, qui peut se réécrire **red** $\cdot\triangleright$ **(A** $\triangleleft$ **prev)** d'après la proposition 1 (**succ**$^{-1}$ étant **prev**). On voit donc progressivement se profiler une définition qui, au terme du procédé, sera celle du champ de données **R** du programme **PREFIX-SUM**. L'ensemble du raisonnement sera détaillé pour un exemple analogue, mais on comprend déjà que ce type de transformation de programmes consiste à combiner plusieurs règles élémentaires.

La construction d'un programme qui raffine un programme donné repose donc sur un *calcul de raffinement* qui stipule que, dans un ensemble d'équations S, si l'équation $E1$ est raffinée par l'équation $E2$ alors, en substituant $E1$ par $E2$ dans S, ce que nous notons $S[E1\backslash E2]$, le programme $P\{S[E1]\}$ est raffiné par $P\{S[E1\backslash E2]\}$. Le raffinement d'une équation lui-même est obtenu :

— soit par l'application d'un changement de base aux deux membres de l'équation :

$$\mathtt{Y} = \mathtt{X} \quad \sqsubseteq \quad \mathtt{h} :: \mathtt{Y} = \mathtt{h} :: \mathtt{X}$$

De plus, les champs égaux `X` et `Y` étant caractérisés par une même paire de fonctions $(v : \sigma)$, si la condition $dom(\sigma) \subseteq dom(\mathtt{h})$ est vérifiée, alors `Y = X` est équivalent à `h :: Y = h :: X` ;

— soit par application d'une règle élémentaire, du type de celle que nous avons vue intuitivement. L'ensemble des règles est présenté dans [VIO 94A]. Nous n'en rappelons que quelques-unes, utilisées dans la suite de l'article.

$$
\begin{array}{llll}
\texttt{Y= X} \lhd \texttt{(g1} \# \texttt{g2)} & \sqsubseteq & \texttt{Y= (X} \lhd \texttt{g1) /\&/ (X} \lhd \texttt{g2)} & (1) \\
\texttt{Y= X} \lhd \texttt{(g1} \circ \texttt{g2)} & \sqsubseteq & \texttt{Y= (X} \lhd \texttt{g1)} \lhd \texttt{g2} & (2) \\
\texttt{Y= f} \rhd \texttt{(X} \lhd \texttt{g)} & \sqsubseteq & \texttt{Y= (f} \rhd \texttt{X)} \lhd \texttt{g} & (3) \\
\texttt{Y= (g1} \# \texttt{g2)} \cdot\rhd \texttt{X} & \sqsubseteq & \texttt{Y= g1} \cdot\rhd \texttt{X /\&/ g2} \cdot\rhd \texttt{X} & (4) \\
\texttt{Y= (g1} \circ \texttt{g2)} \cdot\rhd \texttt{X} & \sqsubseteq & \texttt{Y= g1} \cdot\rhd \texttt{(g2} \cdot\rhd \texttt{X)} & (5)
\end{array}
$$

Ces règles ne définissent pas en général une équivalence, dans la mesure où la validité des expressions déduites suppose des hypothèses supplémentaires. Ainsi, la partie droite de la règle (2) n'est une expression correcte que si ($\texttt{X} \lhd \texttt{g1}$) est un champ de données, ce qui n'est pas requis par la partie gauche de la règle. Par exemple si l'on considère le champ `X` défini sur $dom(\sigma) = dom(v) = [1..n]$ et les fonctions `g1=`λ`(i)|(1<i<=n+1).(i-1)` et `g2=`λ`(i)|(1<=i<=n).(i+1)`, alors ($\texttt{g1} \circ \texttt{g2}$) est l'identité sur $[1..n]$ et la partie gauche de la règle est une expression correcte. Par contre, dans cet exemple, la partie droite serait une expression incorrecte puisque ($\texttt{X} \lhd \texttt{g1}$) n'est pas un champ de données dans la mesure où $dom(v \circ \texttt{g1}) \not\subseteq dom(\sigma)$.

D'autres transformations définissent des équivalences, c'est-à-dire que les expressions transformées définissent des champs de données égaux, sans hypothèse particulière. Par exemple :

$$
\begin{array}{llll}
\texttt{(X1 /\&/ X2)} \lhd \texttt{g} & = & \texttt{(X1} \lhd \texttt{g) /\&/ (X2} \lhd \texttt{g)} & (6) \\
\texttt{h :: (X1/\&/ X2)} & = & \texttt{(h :: X1) /\&/ (h :: X2)} & (7)
\end{array}
$$

3.2. *Stratégies de transformation*

Nous avons vu sur le simple exemple précédent qu'une transformation, qui paraît intuitivement triviale, requiert de combiner soigneusement l'application de plusieurs règles. Dans la pratique il est donc important de proposer des stratégies de transformation, qui montrent comment la combinaison de telles et telles règles élémentaires permet d'obtenir des solutions satisfaisant à certaines caractéristiques données, notamment en matière d'efficacité des programmes. Ici, nous montrons sur l'exemple d'un programme calculant le produit matrice-vecteur comment, par deux transformations successives nous pouvons obtenir, à partir d'une solution ne mettant en œuvre que des communications explicites, un programme plus abstrait utilisant des primitives de communications implantées efficacement dans le modèle de programmation data-parallèle.

Pour montrer l'effet de ces transformations, nous montrons en même temps leur expression sur une version impérative des programmes, écrits en HPF. [3]

3. Un traducteur source-à-source de PEI vers HPF est en cours de développement.

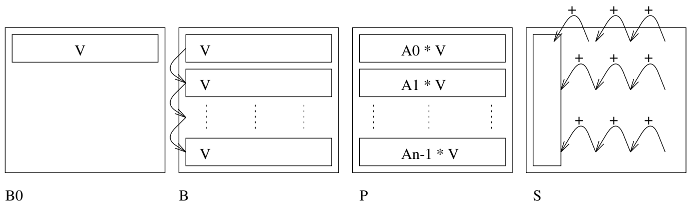

Figure 2. *Produit matrice-vecteur (programme originel)*

3.2.1. *Programme originel*

Le principe de l'algorithme est le suivant : le vecteur `V` est dupliqué sur les lignes de la matrice `A`, pour former en parallèle les produits scalaires (produits termes à termes, puis sommes). Exprimé en PEI, la première version du programme est la suivante : à partir du vecteur `V` en entrée, le champ de données `B0` est défini dans $\mathbb{Z}^2$ avec les valeurs du vecteur `V` sur la ligne $i = 0$. Cette définition est exprimée dans le programme par l'application du changement de base `align` entre `V` et `B0` restreint à cette ligne. Les valeurs de `B0` en cette ligne sont ensuite recopiées récursivement de lignes en lignes pour former le champ de données `B`. La superposition de la matrice `A` et de `B` présente alors les couples nécessaires pour le produit des éléments. Puis la somme est effectuée en additionnant récursivement les colonnes jusqu'à obtenir le résultat final dans la colonne $j = 0$. La figure 2 montre la construction de l'ensemble des champs de données du programme :

$$\texttt{MATVECT1 : (A,V)} \mapsto \texttt{R}$$
$$\left\{\begin{array}{l}
\texttt{A} = \texttt{matrix :: A} \\
\texttt{V} = \texttt{align :: (B0} \lhd \texttt{line)} \\
\texttt{B} = \texttt{B0 /\&/ B} \lhd \texttt{down} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\
\texttt{S} = \texttt{add} \rhd \texttt{(P /\&/ S} \lhd \texttt{pre)} \\
\texttt{R} = \texttt{S} \lhd \texttt{col}
\end{array}\right.$$

$$\begin{array}{lcl}
\texttt{matrix} & = & \lambda\texttt{(i,j)}|\texttt{(0<=i,j<n)} \\
\texttt{line} & = & \lambda\texttt{(i,j)}|\texttt{(i=0)} \\
\texttt{col} & = & \lambda\texttt{(i,j)}|\texttt{(j=0)} \\
\texttt{align} & = & \lambda\texttt{(i,j)}|\texttt{(i=0).(j)} \\
\texttt{down} & = & \lambda\texttt{(i,j)}|\texttt{(0<i<n \& 0<=j<n).(i-1,j)} \\
\texttt{pre} & = & \lambda\texttt{(i,j)}|\texttt{(0<=i<n \& 0<=j<n-1).(i,j+1)} \\
\texttt{prod} & = & \lambda\texttt{(a}\cdot\texttt{b).(a*b)} \\
\texttt{add} & = & \texttt{id} \ \# \ \lambda\texttt{(a}\cdot\texttt{b).(a+b)}
\end{array}$$

La traduction d'un tel énoncé dans un langage impératif data-parallèle suppose de déterminer, s'il existe, un calcul du champ de données en sortie `R` à partir des entrées `A` et `V`. Dans le cas d'une définition récursive comme `B = B0 /&/ B` $\lhd$ `down`, un tel calcul existe si on peut déterminer le point fixe de cette équation. Pour cela considérons l'ensemble des champs de données, muni de la relation d'ordre d'inclusion, dont l'élément minimum est le champ vide ($\perp$). Sur cet exemple, la superposition `B0 /&/ B` $\lhd$ `down`, appliquée récursivement construit une chaîne de cet ensemble ; en dépliant la récurrence on obtient :

$$
\begin{array}{lcl}
B_0 & = & \mathtt{B0} \\
B_1 & = & \mathtt{B0}\ \mathtt{/\&/}\ B_0 \lhd \mathtt{down} \\
\ldots & & \\
B_{n-1} & = & \mathtt{B0}\ \mathtt{/\&/}\ B_0 \lhd \mathtt{down}\ \mathtt{/\&/} \ldots \mathtt{/\&/}\ B_0 \lhd \mathtt{down}^{n-1} \\
B_n & = & \mathtt{B0}\ \mathtt{/\&/}\ B_0 \lhd \mathtt{down}\ \mathtt{/\&/} \ldots \mathtt{/\&/}\ B_0 \lhd \mathtt{down}^{n-1}\ \mathtt{/\&/} \perp = B_{n-1}
\end{array}
$$

Le point fixe cherché est la borne supérieure de cette chaîne, c'est-à-dire le champ B_{n-1}. Cette borne existe parce que le domaine du routage est contraint par un prédicat. La solution peut donc être déterminée par le calcul d'une suite récurrente de terme initial `B0`, calcul qui peut être exprimé par le fragment de code HPF suivant :

```
      FORALL (i=0:n-1,j=0:n-1) B(i,j)=B0(i,j)
      DO k=1,n-1
          FORALL (i=0:n-1,j=0:n-1,(i.GEQ.k))
     *        B(i,j)=B(i-1,j)
      ENDDO
```

Le calcul du champ de données `S` est réalisé en utilisant une technique similaire (lignes 10 à 14 du programme ci-dessous). En effet, à partir de la définition `S = add` $\rhd$ `(P /&/ S` $\lhd$ `pre)`, nous pouvons construire une chaîne de champs de données pour l'inclusion, par restriction du domaine de ces champs. Sur l'exemple, la chaîne est telle que le $k^{\text{ième}}$ champ ne diffère du précédent que par les colonnes $j < k$, d'où l'itération décroissante en k (ligne 12). Nous pouvons donc traduire l'énoncé `MATVECT1` par le programme HPF suivant :

```
 1!HPF$ TEMPLATE T(n,n)
 2!HPF$ ALIGN A(i,j),B(i,j),B0(i,j),P(i,j),S(i,j),R(i,j) WITH T(i,j)
 3!HPF$ ALIGN V(j) WITH B0(0,j)
 4      FORALL (i=0:n-1,j=0:n-1) B(i,j)=B0(i,j)
 5      DO k=1,n-1
 6         FORALL (i=0:n-1,j=0:n-1,(j.GEQ.k))
 7     *        B(i,j)=B(i-1,j)
 8      ENDDO
 9      FORALL (i=0:n-1,j=0:n-1) P(i,j)=A(i,j)*B(i,j)
10      FORALL (i=0:n-1,j=0:n-1) S(i,j)=P(i,j)
11      DO k=n-1,0,-1
12          FORALL (i=0:n-1,j=0:n-1,(i.LT.k))
13     *        S(i,j)=P(i,j)+S(i,j+1)
14      ENDDO
15      FORALL (i=0:n-1,j=0:n-1,j.EQ.0) R(i,j)=S(i,j)
```

En HPF, la directive `TEMPLATE` permet la création d'un domaine géométrique discret sur lequel le programmeur peut aligner les variables parallèles à l'aide de la directive `ALIGN`, c'est-à-dire choisir le placement des valeurs des différentes variables parallèles sur ce domaine discret. En PEI, le domaine potentiel sur lequel sont placées les valeurs d'un champ de données $\mathtt{X}{=}(v : \sigma)$ est $dom(\sigma)$, car $dom(v) \subseteq dom(\sigma)$. La règle générale de traduction consiste donc à exprimer $dom(\sigma)$ par un template dont le domaine parallélépipédique englobe $dom(\sigma)$. Les équations définissant l'égalité des expressions de champ de données, le nombre de points du domaine potentiel est le même pour un ensemble de champs de données mis en relation, et par conséquent un seul template suffit pour représenter les valeurs de ces champs. Ainsi, sur l'exemple, l'équation `A = matrix :: A` précise que `A` a ses valeurs représentées sur le domaine discret $0 \leq i, j < n$ et que la fonction de placement est l'identité, ce qui se traduit par `ALIGN A(i,j) WITH T(i,j)`. Nous déduisons ensuite des équations, que les autres champs ont la même fonction σ_A que le champ `A`, et seront donc traduits par des variables alignées de la même manière que `A` sur le même template. Seul le champ `V` placé sur $\mathbb{Z}$ a une fonction σ_V différente de σ_A ($\sigma_V = \sigma_A \circ \mathtt{align}^{-1}$). La fonction d'alignement n'est donc pas l'identité, mais la fonction `align`, afin que les valeurs de ce champ puisse être placées sur le template `T`.

3.2.2. *Introduction d'une diffusion*

Illustrons maintenant une première stratégie générale de transformation de programmes à partir de cet exemple. La recopie des lignes, dans le programme `MATVECT1`, implique des communications explicites de voisins à voisins. En supposant que l'on dispose d'une primitive de diffusion, caractéristique du modèle de programmation data-parallèle, ces communications peuvent être remplacées par cet opérateur de diffusion. Cela signifie qu'une définition récursive de champ de données (celle de `B` sur l'exemple) est transformée en une définition simple, faisant intervenir une opération de routage, elle-même définie récursivement. Une telle transformation repose sur le résultat suivant, qui décrit une première stratégie de transformation de programmes :

Théorème 1 *Soit* `g` *une fonction définie récursivement par* $\mathtt{g\ =\ i}\ \#\ (\mathtt{g} \circ \mathtt{s})$, *où* `i` *est l'identité définie sur un sous-domaine, et* `s` *est une fonction. L'équation* PEI $\mathtt{Y} = \mathtt{X} \lhd \mathtt{g}$ *est alors raffinée par l'expression de* `Y`*:* $\mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ \mathtt{Y} \lhd \mathtt{s}$

Preuve. Par application des règles de raffinement, nous avons :

$$\begin{array}{lll} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{g} & \\ \equiv & \mathtt{Y} = \mathtt{X} \lhd (\mathtt{i}\ \#\ (\mathtt{g} \circ \mathtt{s})) & \\ \equiv & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ \mathtt{X} \lhd (\mathtt{g} \circ \mathtt{s}) & (1) \\ \sqsubseteq & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{g}) \lhd \mathtt{s} & (2) \end{array}$$

En réitérant le calcul :

$$
\begin{array}{ll}
 & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ (X} \lhd \texttt{(i} \# \texttt{(g} \circ \texttt{s)))} \lhd \texttt{s} \\
\equiv & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ (X} \lhd \texttt{i)} \lhd \texttt{s /\&/ (X} \lhd \texttt{g} \circ \texttt{s)} \lhd \texttt{s} \\
\sqsubseteq & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ X} \lhd \texttt{i} \circ \texttt{s /\&/ X} \lhd \texttt{i} \circ \texttt{s} \circ \texttt{s /\&/} \;..
\end{array}
$$

Déplions maintenant la définition `Y' = X` $\lhd$ `i /&/ Y'` $\lhd$ `s` :

$$
\begin{array}{ll}
 & \texttt{Y'} = \texttt{X} \lhd \texttt{i /\&/ Y'} \lhd \texttt{s} \\
\sqsubseteq & \texttt{Y'} = \texttt{X} \lhd \texttt{i /\&/ X} \lhd \texttt{i} \circ \texttt{s /\&/ (Y'} \lhd \texttt{s)} \lhd \texttt{s} \\
\sqsubseteq & \texttt{Y'} = \texttt{X} \lhd \texttt{i /\&/ X} \lhd \texttt{i} \circ \texttt{s /\&/ X} \lhd \texttt{i} \circ \texttt{s} \circ \texttt{s /\&/} \;..
\end{array}
$$

Pour toute étape n de la récurrence, les n premiers termes des définitions de `Y` et `Y'` sont égaux, donc `Y` = `Y'`. Finalement nous avons :

$$
\begin{array}{ll}
 & \texttt{Y} = \texttt{X} \lhd \texttt{g} \\
\sqsubseteq & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ Y} \lhd \texttt{s}
\end{array}
$$

□

Appliquons ce théorème à l'exemple : la deuxième équation du programme **MATVECT1** est de la même forme que l'expression raffinée de `Y` dans le théorème. On peut donc déduire un nouveau programme PEI, plus abstrait que le précédent, et qui utilise une diffusion :

$$
\texttt{MATVECT2 : (A,V)} \mapsto \texttt{R}
$$

$$
\left\{
\begin{array}{l}
\texttt{A} = \texttt{matrix :: A} \\
\texttt{V} = \texttt{align :: (B0} \lhd \texttt{line)} \\
\texttt{B} = \texttt{B0} \lhd \texttt{spread} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\
\texttt{S} = \texttt{add} \rhd \texttt{(P /\&/ S} \lhd \texttt{pre)} \\
\texttt{R} = \texttt{S} \lhd \texttt{col}
\end{array}
\right.
$$

$$
\texttt{spread} \;=\; \lambda\texttt{(i,j)|(0<=i,j<n).(0,j)}
$$

Preuve. Cet énoncé est obtenu par application du théorème précédent, en réécrivant la fonction `spread` de la manière suivante :

$$
\begin{array}{lcl}
\texttt{spread} & = & \lambda\texttt{(i,j)|(0<=i,j<n).(0,j)} \\
 & = & \lambda\texttt{(i,j)|(i=0)} \;\#\; \lambda\texttt{(i,j)|(0<i<n \& 0<=j<n).(0,j)} \\
 & = & \lambda\texttt{(i,j)|(i=0)} \;\#\; \lambda\texttt{(i,j)|(0<=i,j<n).(0,j) o} \\
 & & \lambda\texttt{(i,j)|(0<i<n \& 0<=j<n).(i-1,j)} \\
 & = & \texttt{line} \;\#\; \texttt{spread o down}
\end{array}
$$

□

La diffusion peut être traduite, en l'état, par une primitive existante dans un langage data-parallèle donné. Cependant, cette instruction implique des communications lors de l'exécution et peut être avantageusement remplacée en HPF par un placement statique des données lors de la phase d'alignement (ligne 3 du programme ci-dessous). Le programme `MATVECT2` peut donc être traduit par le programe HPF suivant :

```
 1!HPF$ TEMPLATE T(n,n)
 2!HPF$ ALIGN A(i,j),B(i,j),P(i,j),S(i,j),R(i,j) WITH T(i,j)
 3!HPF$ ALIGN V(j) WITH B(*,j)
 4      FORALL (i=0:n-1,j=0:n-1) P(i,j)=A(i,j)*B(i,j)
 5      FORALL (i=0:n-1,j=0:n-1) S(i,j)=P(i,j)
 6      DO k=n-1,0,-1
 7          FORALL (i=0:n-1,j=0:n-1,(j.LT.k))
 8     *        S(i,j)=P(i,j)+S(i,j+1)
 9      ENDDO
10      FORALL (i=0:n-1,j=0:n-1,j.EQ.0) R(i,j)=S(i,j)
```

Nous montrons maintenant comment remplacer des communications explicites utilisées lors de la sommation de n nombres, par un opérateur de communications plus abstrait qui est la réduction. Ceci introduit une deuxième stratégie générale de transformation de programmes.

3.2.3. *Introduction d'une réduction*

La seconde partie de l'algorithme précédent consiste à rassembler les produits et à les sommer pour obtenir le vecteur résultat. La définition du champ `S` procède récursivement pour acheminer les produits : les paires de valeurs sont formées par superposition de la matrice `P` et d'elle-même déplacée par l'opération `pre` (cf. figure 3), puis les deux valeurs sont sommées pour n'en rendre qu'une.

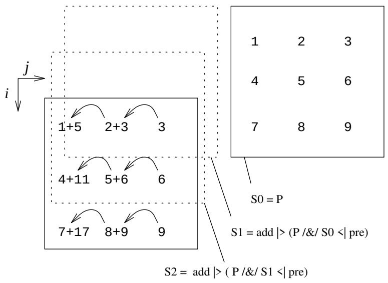

Figure 3. *Routage récursif des produits*

Montrons à présent que la définition récursive du champ `S` peut s'abstraire en une définition non récursive, faisant apparaître une opération de réduction. Cette transformation est validée par l'application du théorème suivant, qui décrit une seconde stratégie de transformation de programmes :

Théorème 2 *Soit* $\texttt{f}$ *une fonction définie par la récurrence* $\texttt{f = i} \# (\texttt{f} \circ \texttt{s}^{-1})$*, où* $\texttt{i}$ *est l'identité définie sur un sous-domaine, et* $\texttt{s}$ *est bijective. L'équation* $\texttt{Y} = \texttt{f} \cdot\triangleright \texttt{X}$ *est alors raffinée par la définition suivante, où le champ intermédiaire* $\texttt{T}$ *est défini récursivement :*

$$\begin{array}{lcl} \texttt{T} & = & \texttt{X /\&/ T} \triangleleft \texttt{s} \\ \texttt{Y} & = & \texttt{T} \triangleleft \texttt{i} \end{array}$$

Preuve. En utilisant les règles de raffinement, nous pouvons écrire :

$$\begin{array}{lll} & \texttt{Y} = \texttt{f} \cdot\triangleright \texttt{X} & \\ \equiv & \texttt{Y} = (\texttt{i} \# (\texttt{f} \circ \texttt{s}^{-1})) \cdot\triangleright \texttt{X} & (def.) \\ \sqsubseteq & \texttt{Y} = \texttt{i} \cdot\triangleright \texttt{X /\&/ } (\texttt{f} \circ \texttt{s}^{-1}) \cdot\triangleright \texttt{X} & (4) \\ \sqsubseteq & \texttt{Y} = \texttt{i} \cdot\triangleright \texttt{X /\&/ f} \cdot\triangleright (\texttt{s}^{-1} \cdot\triangleright \texttt{X}) & (5) \\ \equiv & \texttt{Y} = \texttt{X} \triangleleft \texttt{i /\&/ f} \cdot\triangleright (\texttt{X} \triangleleft \texttt{s}) & (prop.1) \end{array}$$

En itérant le calcul :

$$\begin{array}{lll} \sqsubseteq & \texttt{Y} = \texttt{X} \triangleleft \texttt{i /\&/ } (\texttt{X} \triangleleft \texttt{s}) \triangleleft \texttt{i /\&/ f} \cdot\triangleright ((\texttt{X} \triangleleft \texttt{s}) \triangleleft \texttt{s}) & \\ \equiv & \texttt{Y} = (\texttt{X /\&/ X} \triangleleft \texttt{s /\&/ } (\texttt{X} \triangleleft \texttt{s}) \triangleleft \texttt{s /\&/ } \ldots) \triangleleft \texttt{i} & (6) \\ \equiv & \texttt{Y} = \texttt{T} \triangleleft \texttt{i} \; avec \; \texttt{T} = \texttt{X/\&/T} \triangleleft \texttt{s} & \end{array}$$

□

Nous pouvons maintenant écrire le programme PEI suivant, qui abstrait le programme `MATVECT2` :

$$\texttt{MATVECT3 : (A,V)} \mapsto \texttt{R}$$
$$\left\{ \begin{array}{l} \texttt{A} = \texttt{matrix :: A} \\ \texttt{V} = \texttt{align :: (B0} \triangleleft \texttt{line)} \\ \texttt{B} = \texttt{B0} \triangleleft \texttt{spread} \\ \texttt{P} = \texttt{prod} \triangleright \texttt{(A /\&/ B)} \\ \texttt{S} = \texttt{sum} \triangleright \texttt{(red} \cdot\triangleright \texttt{P)} \\ \texttt{R} = \texttt{S} \triangleleft \texttt{col} \end{array} \right.$$
$$\texttt{red} = \lambda\texttt{(i,j)|(0<=i,j<n).(i,0)}$$
$$\texttt{sum} = \texttt{id} \; \# \; \lambda\texttt{(a}\cdot\texttt{b).(a+sum(b))}$$

Preuve. Nous pouvons réécrire la fonction `red` et obtenir une définition semblable à celle de `f` dans le théorème précédent :

$$\begin{array}{lcl} \texttt{red} & = & \lambda\texttt{(i,j)|(0<=i<n \& j=0).(i,0)} \; \# \\ & & \lambda\texttt{(i,j)|(0<=i<n \& 0<j<n).(i,0)} \\ & = & \texttt{col} \# \lambda\texttt{(i,j)|(0<=i<n \& 0<=j<n).(i,0) o} \\ & & \lambda\texttt{(i,j)|(0<=i<n \& 0<j<n).(i,j-1)} \\ & = & \texttt{col} \# \texttt{red o } (\lambda\texttt{(i,j)|(0<=i,j<n-1).(i,j+1)})^{-1} \\ & = & \texttt{col} \# \texttt{red o (pre)}^{-1} \end{array}$$

Posons $\texttt{Y = red} \cdot\triangleright \texttt{P}$. On déduit du thérorème précédent, un raffinement de la définition de `Y` sous la forme :

$$\left\{ \begin{array}{l} \texttt{T} = \texttt{P/\&/T} \triangleleft \texttt{pre} \\ \texttt{Y} = \texttt{T} \triangleleft \texttt{col} \end{array} \right.$$

L'équation $\texttt{S = sum} \rhd \texttt{Y}$ de $\texttt{MATVECT3}$ devient donc $\texttt{S = sum} \rhd \texttt{(T} \lhd \texttt{col)}$. Cette définition de `S` peut se réécrire sous une forme équivalente en utilisant la règle (3), ce qui donne $\texttt{S = (sum} \rhd \texttt{T)} \lhd \texttt{col}$.

Il nous reste à utiliser le fait que la somme `sum` d'une séquence de longueur quelconque peut se réécrire en fonction de la somme binaire `add`, de la façon suivante :

$$\texttt{sum = id} \# \lambda\texttt{(a}\cdot\texttt{b).(add(a}\cdot\texttt{sum(b)))}$$

Utilisons maintenant la proposition suivante :

Proposition 2. *Soit une fonction récursive* $\texttt{f = id} \; \# \lambda\texttt{(a}\cdot\texttt{b).(a} \otimes \texttt{f(b))}$, *où* $\otimes$ *est un opérateur binaire associatif. Alors on a :*

$$\texttt{f} \rhd \texttt{(X1 /\&/ X2)} = \texttt{f} \rhd \texttt{(f} \rhd \texttt{X1 /\&/ f} \rhd \texttt{X2)}$$

$\diamond$

Alors
$$\begin{aligned} \texttt{sum} \rhd \texttt{T} &= \texttt{sum} \rhd \texttt{(P /\&/ T} \lhd \texttt{pre)} \\ &= \texttt{sum} \rhd \texttt{(sum} \rhd \texttt{P /\&/ sum} \rhd \texttt{(T} \lhd \texttt{pre))} \end{aligned}$$

Puisque le champ de données `P` n'est formé que de séquences d'un élément, on peut déduire : $\texttt{sum} \rhd \texttt{P} = \texttt{P}$. De même, puisque $\texttt{(P /\&/ sum} \rhd \texttt{(T} \lhd \texttt{pre))}$ est formé de séquences d'au plus deux éléments, on a :

$$\texttt{sum} \rhd \texttt{T} \;=\; \texttt{add} \rhd \texttt{(P /\&/ (sum} \rhd \texttt{T)} \lhd \texttt{pre)}$$

En renommant $\texttt{sum} \rhd \texttt{T}$ en `S'`, nous avons $\texttt{S'} = \texttt{add} \rhd \texttt{(P /\&/ S'} \lhd \texttt{pre)}$

De plus, comme `col o col = col`, nous avons :

$$\begin{aligned} \texttt{R} &= \texttt{((sum} \rhd \texttt{T)} \lhd \texttt{col)} \lhd \texttt{col} \\ &= \texttt{(sum} \rhd \texttt{T)} \lhd \texttt{col} \\ &= \texttt{S'} \lhd \texttt{col} \end{aligned}$$

Au final, nous avons $\begin{cases} \texttt{S'} = \texttt{add} \rhd \texttt{(P /\&/ S'} \lhd \texttt{pre)} \\ \texttt{R} = \texttt{S'} \lhd \texttt{col} \end{cases}$

Ces deux dernières équations sont présentes dans `MATVECT2` et nous avons montré qu'elles sont un raffinement de l'équation définissant `R` de `MATVECT3`. Les deux programmes ne différant que par ces équations, nous concluons que `MATVECT3` abstrait `MATVECT2`.

$\square$

Note. La démonstration eût été plus simple si nous avions également utilisé la fonction `sum` dans les programmes précédents, fonction qui aurait été appliquée sur la séquence finale obtenue par superpositions successives. Cependant, ce programme n'aurait pu être traduit tel quel dans un langage data-parallèle impératif en raison de l'absence du type liste.

Nous obtenons ainsi un programme utilisant l'opérateur de réduction disponible dans le modèle de programmation data-parallèle. Le choix du chemin pour la réduction est cette fois-ci laissé au compilateur, sous l'hypothèse qu'il utilisera le chemin optimal pour une machine donnée.

```
!HPF$ TEMPLATE T(n,n)
!HPF$ ALIGN A(i,j),B(i,j),P(i,j),S(i,j),R(i,j) WITH T(i,j)
!HPF$ ALIGN V(j) WITH B(*,j)
      FORALL (i=0:n-1,j=0:n-1) P(i,j)=A(i,j)*B(i,j)
      FORALL (i=0:n-1,j=0:n-1) S(i,0)=SUM(P(i,j))
      FORALL (i=0:n-1,j=0:n-1,j.EQ.0) R(i,j)=S(i,j)
```

4. Conclusion

Les concepts fondamentaux de la programmation data-parallèle sont l'expression d'opérations globales sur des variables dites parallèles. L'alignement initial des variables entre elles, leur distribution, et celle des calculs afférents, sur un ensemble de processeurs virtuels, les communications induites, sont autant de questions essentielles pour l'écriture de programmes efficaces. C'est dire l'intérêt, dans ce contexte, d'un cadre formel permettant de valider des transformations de programmes. Ces transformations peuvent servir, de manière générale, à obtenir un programme utilisant des primitives dont on sait qu'elles sont implantées de manière efficace dans le langage cible.

Pei offre ce cadre et propose un calcul de raffinement adéquat. Dans cet article, nous avons montré quelques résultats permettant de définir des stratégies de transformations, portant sur des fragments de programmes qui traitent des communications, en vue des caractéristiques visées.

Pour mener à bien et faciliter l'écriture et la transformation des programmes, un environnement permettant d'appliquer interactivement les règles de transformations, ainsi qu'un outil de construction graphique de ces programmes sont disponibles. Par ailleurs, un traducteur Pei-Caml, fondé sur l'interprétation fonctionnelle des champs de données, permet d'exécuter les programmes obtenus. Enfin, un prototype de traducteur Pei-HPF a été réalisé et permet de traduire un sous-ensemble des programmes Pei, sur la base des mécanismes de traduction esquissés dans l'article, vers le langage data-parallèle HPF. Ces programmes sont supposés ne comporter actuellement que des définitions de champs de données déterministes et explicites. Ceci implique que les séquences de valeurs formées par une réduction géométrique doivent être réduites par un opérateur associatif et commutatif, et que chaque équation possède un membre réduit à un champ de données simple, ce qui permet d'assimiler les équations à des affectations. En pratique, ces hypothèses sur les programmes Pei n'impliquent pas de restriction vis à vis de la programmation HPF.

5. Bibliographie

[BAC 88] Back R.J., « A calculus of refinements for program derivations », *Acta Informatica*, 25, pp. 593–624, 1988.

[BOU 94] Bougé L., Le Guyadec Y., Utard G., Virot B., « A proof system for a simple data-parallel programming language », *Proc. IFIP WG 10.3 Conf. on Applications in Parallel and Distributed Systems*, Caracas, Avril 1994.

[CHE 91] Chen M., Choo Y., Li J., *Parallel Functional Languages and Compilers*, Frontier Series, ACM Press, Chapitre 7, 1991.

[HIG 93] *High Performance Fortran Language Specification*, Version 1.0, Mai 1993.

[KRU 85] Kruskal C.P., Rudolph L., Snir M., « The Power of Parallel Prefix », *IEEE Transactions on computers*, vol. C-34, No. 10, p. 965–968, Octobre 1985.

[MAU 89] Mauras C., Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones, Thèse Université de Rennes I, 1989.

[MOR 90] Morgan C., *Programming from specifications*, C.A.R. Hoare, Prentice Hall Ed., N.J., 1990.

[TSI 93] Bougé L. et al., *Langages à parallélisme de données, Technique et Science Informatiques*, 12(6), Septembre 1993.

[VIO 92] Violard E., Une théorie unificatrice pour la construction de programmes parallèles par des techniques de transformations, Thèse Université de Franche-Comté, Octobre 1992.

[VIO 94A] Violard E., Perrin G.R., « Reduction in PEI », *CONPAR'94*, LNCS 854, p. 112–123, 1994.

[VIO 94B] Violard E., « A mathematical theory and its environment for parallel programming », *Parallel Processing Letters*, 4, p. 313-328, 1994.