Data-aware Process Networks

Christophe Alias, Alexandru Plesco

Data-aware Process Networks

Christophe Alias, Alexandru Plesco

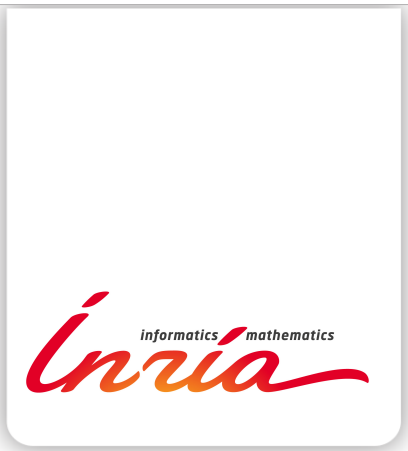
RESEARCH REPORT

N° 8735

June 2015

Project-Team Compsys

ISSN 0249-6399 ISRN INRIA/RR--8735--FR+ENG



Data-aware Process Networks

Christophe Alias*, Alexandru Plesco†

Project-Team Compsys

Research Report n° 8735 — June 2015 — 32 pages

Abstract: This report presents the Data-aware Process Networks, a new parallel execution model adapted to the hardware constraints of high-level synthesis, where the data transferts are made explicit. We show that the DPN model is consistant in the meaning where any translation of a sequential program produces an equivalent DPN without deadlocks. Finally, we show how to compile a sequential program to a DPN and how to optimize the input/output and the parallelism.

Key-words: High-level Synthesis, automatic parallelization, I/O optimization, polyhedral compilation

* Inria/ENS-Lyon/UCBL/CNRS
† The XtremLogic™ company

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Data-aware Process Networks

Résumé : Ce rapport présente les réseaux de processus DPN (Data-aware Process Network), un modèle d'exécution paralléle adapté aux contraintes matérielles de la synthèse de circuit dans lequel les transferts de données avec une mémoire distante sont explicites. Nous montrons que le modèle DPN est consistant dans le sens où toute traduction d'un programme séquentiel en DPN produit un DPN équivalent et garanti sans interblocage. Enfin, nous expliquons comment compiler un programme séquentiel vers un DPN, et comment optimiser les entrées/sorties et le parallélisme

Mots-clés : Synthèse de circuits haut-niveau, parallélisation automatique, optimisation des entrées/sorties, compilation polyédrique

1 Introduction

Les réseaux de processus sont des modèles d'exécution qui expriment naturellement le parallélisme d'un calcul. A ce titre, ils sont étudiés en tant que modéles de systèmes parallèles, et permettent de construire des études théoriques, des analyses et des mesures sur les différentes problématiques liées au parallélisme (ordonnancement, équilibrage de charge, allocation, checkpointing, consommation, etc). De même, ils constituent la base des paradigmes de programmation paralléle, notamment pour les langages de type streaming [11] où ils sont manipulés explicitement; mais aussi pour les langages de type PGAS (Partitionned Global Array Space), où les tÃ¢ches peuvent être vues comme des processus qui communiquent par les tableaux globaux [10, 6]. Enfin, les réseaux de processus constituent une représentation intermédiaire naturelle pour un compilateur paralléliseur, dans lequel le front-end extrait le parallélisme et produit un réseau de processus, et le back-end compile le réseau de processus vers l'architecture cible.

La synthèse de circuits (HLS, high-level synthesis) consiste à compiler un programme écrit dans un langage de haut-niveau (comme C) en circuit. Le circuit doit être le plus efficace possible tout en utilisant au mieux les ressources disponibles (consommation, surface de silicium, unités LUT sur FPGA, accés mémoire, etc). Si de nombreuses avancées ont été réalisées sur les aspects back-end (construction des pipelines, placement/routage), les aspects front-end (parallélisme, entrées/sorties) restent encore rudimentaires, et très en deçà de ce qui existe dans la communauté calcul haute-performance (HPC, high-performance computing), notamment dans le modèle polyédrique [8].

Dans ce rapport, nous présentons un modèle de réseaux de processus adapté aux contraintes de la HLS, le modèle DPN (Data-aware Process Network). Le modèle DPN explicite les communications avec la mémoire centrale, les accès parallèles aux canaux de communication, et est suffisament proche des contraintes matérielles pour pouvoir être traduit directement en circuit. Nous montrons comment compiler un programme impératif en réseau DPN, de façon à optimiser à la fois les entrées/sorties et le parallélisme, en utilisant le formalisme du modèle polyédrique.

Ce rapport est structuré de la façon suivante. La Section 2 introduit le modèle polyédrique et les réseaux de processus réguliers. Ensuite, la Section 3 défini les réseaux DPN et montre leur déterminisme et l'absence d'interblocage. La Section 4 explique comment compiler un programme impératif en DPN et montre comment optimiser les entrées/sorties et le parallélisme. La Section 5 présente les travaux liés et montre leur relation avec les DPN. Enfin, la Section 6 conclue ce rapport et présente les travaux futurs.

2 Préliminaires

Cette section défini les concepts et les notations utilisés dans ce rapport. Les sous-sections 2.1, 2.2 et 2.3 présentent le modèle polyédrique, le formalisme dans lequel sont écrites toutes les analyses présentées dans ce rapport. Ensuite, la sous-section 2.4 présente les réseaux de processus réguliers et introduit les notions d'ordonnancement observable et de schéma de compilation, qui seront utilisées pour prouver la correction des DPN.

2.1 Modèle polyédrique

Le modèle polyèdrique est un cadre de conception d'analyses exactes et d'optimisations sur des programmes à *contrôle statique*. C'est à dire des programmes qui ne manipulent que des tableaux statiques et des variables de type simple (pas de pointeurs) et dont le contrôle est limité aux boucles `for` et aux `if`. Une caractéristique essentielle est que les fonctions d'indice des tableaux, les bornes des boucles et les tests des `if` sont des *fonctions affines* des compteurs des boucles `for`

englobantes. Le *vecteur d'itération veci* d'une affectation S est formé des compteurs des boucles qui l'englobe. L'instance d'exécution $(S, \vec{i})$ est appelée *opération*. Les opérations sont les unités de base des analyses polyédriques. Le *domaine d'itération* d'une affectation est l'ensemble de tous ses vecteurs d'itération au cours de l'exécution. Sous ces restrictions (boucles `for`, `if`, contraintes affines), le domaine d'itération d'une affectation est exactement un *polyèdre convexe*, dont la forme ne dépend pas des entrées. On peut alors le calculer statiquement, et appliquer diverses opérations polyédriques (opérations géométriques, optimisation linéaire, comptage, etc) pour construire des analyses statiques. De nombreuses analyses et transformations de programmes ont ainsi été définies dans ce cadre, qui existe depuis les années 90.

Dans ce rapport, on distinguera dans chaque opération $W = f(R_1, \ldots, R_n)$ les parties lecture (R_1, ..., R_n) et écriture (W). Pour un programme à contrôle statique P, on note $\Omega(P)$ l'ensemble des *opérations élémentaires* ainsi obtenu. On note $\prec_{\text{seq}}$ l'ordre d'exécution séquentiel sur $\Omega(P)$. On convient que les lectures R_i d'une opération sont exécutées avant l'écriture: $R_i \prec_{\text{seq}} W, \forall i$. Enfin, on note $\mathcal{M}(P)$ l'ensemble des emplacements mémoire accédés par chaque opération élémentaire $X \in \Omega(P)$. On note $\mu(X)$ l'emplacement accédé (en lecture ou en écriture) par X), $\mu : \Omega(P) \to \mathcal{M}(P)$. Par commodité, dans toute la suite, on dira *opération* pour *opération élémentaire*.

2.2 Dépendances, ordonnancement

Il existe une *dépendance* $X \to Y$ entre deux opérations $X, Y \in \Omega(P)$ lorsque X s'exécute avant Y ($X \prec_{\text{seq}} Y$) et lorsque X et Y accèdent au même emplacement mémoire ($\mu(X) = \mu(Y)$). Selon la nature de X et de Y (lecture ou écriture), on parle de dépendance de *flot* (W puis R), *anti* (R puis W) ou *sortie* (W puis W). Les dépendances anti et sortie n'expriment que des conflits d'accès à la mémoire et peuvent être supprimés en modifiant l'utilisation de la mémoire. En réalité, seules comptent les dépendances de flot, qui expriment à leur façon le calcul réalisé par le programme. Plus précisement, on s'interesse aux dépendances de flot $W \to R$ ou W est le dernière écriture de $\mu(R)$ qui précéde R. En somme, le W qui défini la valeur lue par R. Ce W, qui est unique, est appelé *source* de R. On le note $s(R)$. On a:

$$s(R) = \max_{\prec_{\text{seq}}} \{W \in O(P),\ \mu(W) = \mu(R) \wedge W \prec_{\text{seq}} R\}$$

Un *ordonnancement* est une application θ qui associe à chaque opération du programme une date d'exécution appartenant à un ensemble totalement ordonné, $\theta : \Omega(P) \to (\mathcal{T}, \ll)$. La plupart des transformations de programmes (optimisations, etc) peuvent d'écrire avec une fonction d'ordonnancement. Le modèle polyédrique fourni des techniques pour calculer des ordonnancements affines $\theta(S, \vec{i}) = A\vec{i} + \vec{b} \in (\mathbb{N}^p, \ll)$, ou $\ll$ désigne l'*ordre lexicographique*. Un ordonnancement est *correct* s'il respecte la causalité des dépendances: $X \to Y \Rightarrow \theta(X) \ll \theta(Y)$. L'ordre d'exécution induit par θ est noté $\prec_\theta$. On défini s_θ en remplaçant *mutatis mutandis* $\prec_{\text{seq}}$ par $\prec_\theta$ dans la relation ci-dessus. On a la propriété suivante, qui garanti que les valeurs calculées sont bien les bonnes:

Propriété 1 (Causalité) *Un ordonnancement θ est correct si et seulement si $s = s_\theta$.*

Exemple. Le programme donné en figure 1.(a) est un programme à contrôle statique qui effectue un calcul de relaxation binomiale itératif. Les bords du tableau a sont initialisés (affectations I_1 et I_2), ensuite, on calcule itérativement une moyenne sur chaque fenêtre de trois cases (affectations S et T). Enfin, on récupére le résultat (affectation R).

Les domaines d'itération des affectations sont donnés en (b). Les domaines des affectations I_1, I_2 et R sont réduits à un point. Les domaines des affectations S et T sont des rectangles,

qu'on superpose ici pour faciliter la présentation. Les points noirs (•) réprésentent les itérations de S, et les points bleus (•) représentent les itérations de T. Ce programme comporte tous les types de dépendances, dont quelques instances sont représentées par des fléches sur le dessin.

- Dépendances de *flot*: $(I_1,) \rightarrow (S,t,0)$, $(I_2,) \rightarrow (S,t,N-2)$, $(S,t,i) \rightarrow (T,t,i)$, $(T,t,i) \rightarrow (S,t+1,i-1)$, $(T,t,i) \rightarrow (S,t+1,i)$, $(T,t,i) \rightarrow (S,t+1,i+1)$, $(T,K-1,1) \rightarrow (R,)$.
- Dépendances *anti* (en pointillés): $(S,t,i) \rightarrow (T,t,i-1)$, $(S,t,i) \rightarrow (T,t,i)$, $(S,t,i) \rightarrow (T,t,i+1)$, $(T,t,i) \rightarrow (S,t+1,i)$
- Dépendances de *sortie*: $(S,t,i) \rightarrow (S,t+1,i)$ $(T,T-1,1) \rightarrow (R,)$, toutes ces dépendances sont inclues dans les dépendances de flot, elles ne sont donc pas représentées.

Un ordonnancement valide est $\theta(I_1,) = \theta(I_2,) = (0)$, $\theta(S,t,i) = (1,2t+i,t,0)$, $\theta(T,t,i) = (1,2t+i+1,t,1)$, $\theta(R,) = (2)$. L'ordre d'exécution correspondant est le suivant. On exécute d'abord I_1 et I_2 en même temps. Ensuite, on exécute les instances de S et T de façon entrelaçée, comme indiqué sur le dessin (flèches vertes). Enfin, on exécute R. On peut vérifier que cet ordonnancement est valide, en ce sens qu'il satisfait bien toutes les dépendances. Par exemple, la première dépendance anti est respectée puisque $\theta(S,t,i) = (1,2t+i,t,0) \ll \theta(T,t,i-1) = (1,2t+i-1+1,t,1) = (1,2t+i,t,1)$. □

2.3 Allocation mémoire, assignation unique

Une *allocation mémoire* est une application σ qui associe à chaque emplacement mémoire $m \in \mathcal{M}(P)$ adressé par P, un emplacement alternatif $\sigma(m)$ dans un nouvel ensemble d'emplacement mémoire $\mathcal{M}'$, $\sigma : \mathcal{M}(P) \rightarrow \mathcal{M}'$. Etant fixé un ordonnancement θ, deux emplacements mémoire m et $m' \in CM(P)$ *θ-interfèrent* s'il existe $W_1, W_2, R_1, R_2 \in CM(P)$ avec $\mu(W_1) = \mu(R_1) = m$, $\mu(W_2) = \mu(R_2) = m'$, $\theta(W_1) \ll \theta(R_2)$, et $\theta(W_2) \ll \theta(R_1)$. On note alors $m \bowtie_\theta m'$. Une allocation mémoire σ est *θ-correcte* si:

$$m \bowtie_\theta m' \Rightarrow \sigma(m) \neq \sigma(m')$$

Un *programme transformé* est entièrement décrit par la donnée: (i) des opérations du programme $\Omega(P)$, d'un ordonnancement valide θ, et d'une allocation θ-correcte σ: $(\Omega(P), \theta, \sigma)$. Le programme initial peut s'écrire $(\Omega(P), \theta_{\text{seq}}, Id)$, et on a l'équivalence (de calcul): $(\Omega(P), \theta_{\text{seq}}, Id) = (\Omega(P), \theta, \sigma)$.

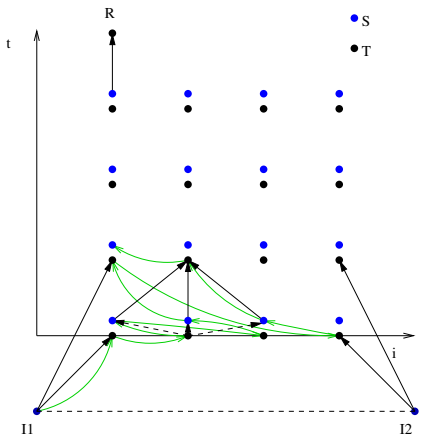
Un programme est en *assignation unique* si chaque opération d'écriture écrit un emplacement mémoire différent, autrement dit si pour $W_1, W_2 \in \Omega(P), \sigma(W_1) = \sigma(W_2) \Rightarrow W_1 = W_2$. Pour passer un programme en assignation unique, on se sert de la fonction source s pour lire les bons emplacements. On forme $\mathcal{M}' = \{W,\ W \in \Omega(P)\}$ et on défini σ par $\sigma(W) = W$ pour chaque écriture W de $\Omega(P)$ et $\sigma(R) = s(R)$ pour chaque lecture. Le programme transformé $(\Omega(P), \theta, \sigma)$ est alors en assignation unique.

Exemple (suite). Pour passer le programme Jacobi 1D en assignation unique, on commence par calculer la fonction source $s(.)$.

- $s((I_1,) : a[0]) = a[0]$, la source de la lecture a[0] de l'opération $(I_1,)$ n'existe pas dans le programme, on garde donc la valeur en entrée de $a[0]$.
- $s((I_2,) : a[N-1]) = a[N-1]$, pour les mêmes raisons.

```
I1 :  a[0] = 0;
I2 :  a[N-1] = 0;
      for(t=0; t ≤ K-1; t++)
        {
        for(i=1; i ≤ N-2; i++)
S :       b[i] = a[i-1] + a[i] + a[i+1];
        for(i=1; i ≤ N-2; i++)
T :       a[i] = b[i];
        }
R :   result = a[1];
```

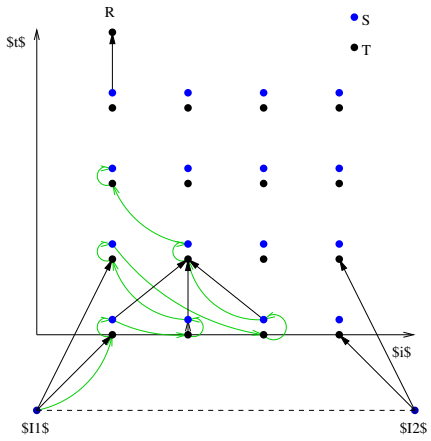
(a) Jacobi 1D



(b) Domaines d'iteration + dépendances

```
I1 :  I1 = 0;
I2 :  I2 = 0;
      for(t=0; t ≤ K-1; t++)
        {
        for(i=1; i ≤ N-2; i++)
S :       S'[t%1][i%1] =
            (i = 1 ? I1 : (t=0 ? a[i-1] : T[t-1][i-1])) +
            (t = 0 ? a[i] : T[t-1][i]) +
            (i = N − 2 ? I2 : (t=0 ? a[i+1] : T[t-1][i+1]));
        for(i=1; i ≤ N-2; i++)
T :       T[t][i] = S'[t%1][i%1];
        }
R :   R = T[K-1][1];
```

(c) Jacobi 1D, assignation unique



(d) Ordre d'exécution

Figure 1: Jacobi 1D: itérations, dépendances, ordonnancement

- Pour les lectures de (S,t,i), on obtient les sources suivantes:

$$
\begin{aligned}
s\left((S,t,i):a[i-1]\right) &= \begin{cases} i=1: & (I_1,) \\ i\geq 2 \wedge t=0: & a[i-1] \\ i\geq 2 \wedge t\geq 1: & T[t-1][i-1] \end{cases} \\
s((S,t,i):a[i]) &= \begin{cases} t=0: & a[i] \\ t\geq 1: & T[t-1][i] \end{cases} \\
s((S,t,i):a[i+1]) &= \begin{cases} i=N-2: & (I_2,) \\ i\leq N-2 \wedge t=0: & a[i+1] \\ i\leq N-2 \wedge t\geq 1: & T[t-1][i+1] \end{cases}
\end{aligned}
$$

Sur les bords ($i=1$ ou $i=N-2$) la source est donnée par une affectation d'initialisation (I_1 ou I_2). Quand $t=0$, la source est la valeur initiale du tableau (comme au premier item). Autrement, la valeur est produite par une instance de T ($(T,t-1,i-1)$, ou $(T,t-1,i)$ ou $(T,t-1,i+1)$).

On obtient alors les lectures du programme en assignation unique donné en (c). Tel quel (avant avoir appliqué l'allocation $(t,i) \mapsto (t\%1, i\%1)$ à S), le programme ne comporte plus que des dépendances de flot, et peut donc être ordonnancé avec $\theta(S,t,i) = (1, t+i, t, 0)$, $\theta(S,t,i) = (1, t+i, t, 1)$. L'ordonnancement de I_1, I_2 et R reste inchangé. L'ordre d'exécution est décrit sur dessin en (d) (flèches vertes). Les intervalles de vie des cases $S[t][i]$ sont disjoints, et donc, on peut appliquer à S l'allocation $\sigma(S[t][i]) = S'[t\%1][i\%1]$. L'onglet (c) représente le programme transformée (sans suivre l'ordre de θ). □

2.4 Réseaux de processus réguliers

Un *réseau de processus réguliers* R est la donnée des éléments suivants:

- $\mathcal{P}$, un ensemble de *processus*.
- $\mathcal{C} = \{C_1, \dots C_n\}$, un ensemble de mémoires appelées *canaux*.
- Un ensemble de *connexions* de $\mathcal{P} \times \mathcal{M} \times \mathcal{P}$.
- Pour chaque *canal* $C \in \mathcal{C}$, l'historique des écritures $\mathcal{W}(C) = W_1^C, \dots, W_p^C$ et l'historique des lectures $\mathcal{R}(C) = R_1^C, \dots, R_p^C$ sont *invariants*. Cette contrainte implique que la trace d'exécution de chaque processus est fixe. En particulier, les processus peuvent être décrits, mais pas seulement, par des programmes à contrôle statique.
- Un *ordre de précédence* entre écritures et lectures $\prec_{\text{prec}} \subset \mathcal{W}(C) \times \mathcal{R}(R)$. $\prec_{\text{prec}}$ est étendu pour exprimer l'ordre séquentiel des écritures ($W_i^C \prec_{\text{prec}} W_j^C$ si $i < j$) et des lectures ($R_i^C \prec_{\text{prec}} R_j^C$ si $i < j$) sur chaque canal C. Dans la suite, on note $\Omega(R)$ l'ensemble des opérations exécutées par le réseau R, $\Omega(R) = \cup_{C \in \mathcal{C}} \mathcal{R}(C) \cup \mathcal{W}(C)$.

Un ordonnancement θ_{obs} des opérations de R est *observable* s'il décrit une exécution possible des opérations de R, autrement dit si pour $X, Y \in \Omega(R)$, $X \prec_{\text{prec}} Y \Rightarrow \theta_{\text{obs}}(X) \ll \theta_{\text{obs}}(Y)$.

La *compilation* d'un programme P à contrôle statique en réseau de processus régulier est la donnée des éléments suivants:

- Un *placement* $\Pi : \Omega(P) \to \mathcal{P}$
- Une *allocation des canaux* $\sigma : \Omega(P) \to \cup_{C_i \in \mathcal{C}} C_i$
- Un ensemble de *synchronisations* qui garantissent un ordre de précédence $\prec_{\text{synchro}} \subset \Omega(P) \times \Omega(P)$ total sur chaque processus: $\Pi(X) = \Pi(Y) \Rightarrow X \prec_{\text{synchro}} Y ou Y \prec_{\text{prec}} X$ et le respect des dépendances de flot: $W = s(R) \Rightarrow W \prec_{\text{prec}} R$.

Un réseau est *partiellement équivalent à* P si toutes ses exécutions qui terminent produisent le même calcul que P, c'est à dire si tous les ordonnancements observables dans R totalement définis sur $\Omega(R)$ sont des ordonnancements corrects de P. Si en plus le réseau est garanti sans interblocage, on dit qu'il est *équivalent* à P et que la compilation est *correcte*.

3 Data-aware Process Networks

Cette section présente les réseaux DPN, formalise la traduction (compilation) d'un programme impératif en DPN, et montre qu'un DPN compilé est toujours équivalent au programme (théorème 1), et qu'il est garanti sans interblocage (théorème 2).

Un *data-aware process network (DPN)* est un réseau de processus régulier qui vérifie les conditions suivantes.

- Chaque canal admet un *unique* processus source et un *unique* processus destination.
- Les canaux sont des mémoires *bloquantes* en écriture et en lecture.

Un schéma de compilation $(\Pi, \sigma, \prec_{\text{prec}})$ d'un programme à contrôle statique P vers un DPN doit vérifier les conditions suivantes:

- Pour chaque lecture dans un canal, la source est bien écrite dans le même canal: si (P_1, C, P_2) est une connexion de P_1 vers P_2 en passant par le canal C, alors $\Pi(R) = P_2 \wedge \sigma(R) \in C \Rightarrow \Pi(s(R)) = P_1 \wedge \sigma(s(R)) \in C$.
- On choisi un ordonnancement correct θ dont l'ordre induit sur $\Omega(P)$ est total, et on choisi une allocation σ qui est θ-correcte.
- L'ordre de précédence est total sur chaque processus, et non défini partout ailleurs, $\prec_{\text{prec}} = \prec_\theta \cap \{(X, Y),\ \Pi(X) = \Pi(Y)\}$.
- Pour chaque écriture W, la dernière lecture $d(W) = \max_{\prec_{\text{prec}}}\{R,\ s(R) = W\}$ *libère* la case écrite par W, $\sigma(W)$. La case $\sigma(W)$ doit alors être à nouveau écrite pour pouvoir être lue.

Pour montrer que la compilation est correcte, on montre que le DPN obtenu est partiellement équivalent au programme original P (théorème 1), et ensuite qu'il ne se produit jamais d'interblocage (théorème 2).

Théorème 1 (Equivalence partielle) *La compilation de P par le schéma $(\Pi, \sigma, \prec_{prec})$ décrit ci-dessus produit un DPN R partiellement équivalent à P.*

Preuve. Soit θ_{obs} un ordonnancement observable de R. Il suffit de montrer qu'il est correct dans P, c'est à dire que $s_{\text{obs}} = s$. Considérons un canal C d'un processus P_1 vers un processus P_2, et un emplacement $e \in C$. notons $\mathcal{W}(C[e]) = W_1^e, \ldots, W_k^e$ l'historique des écritures de $\mathcal{W}(C)$ dans e et $\mathcal{R}(C[e]) = R_1^e, \ldots, R_\ell^e$ l'historique des lectures de $\mathcal{R}(C)$ dans e. Soit $u \in (\mathcal{W}(C[e]) \cup \mathcal{R}(C[e]))^*$ la trace d'exécution des opérations d'accés à e (vérifiant θ_{obs}). u commence par une écriture (lecture bloquante), et on n'a jamais deux écritures consécutives (écriture bloquante) u est donc de la forme $W_1^e \text{Reads}_1 W_2^e \mathit{Reads}_2 \ldots W_k^e \mathit{Reads}_k$ avec $\text{Reads}_1 \ldots \text{Reads}_k = R_1^e \ldots . R_\ell^e$.

Il suffit de montrer que pour chaque i, $i = 1, k$, pour chaque $R \in \text{Read}_i$, on a $W_i = s(R)$. Autrement dit, que R "lit bien la bonne valeur". La suite de la preuve comporte donc deux parties:

1. On montre que les lectures de Read_i partagent la même source.
2. On montre ensuite que cette source est W_i.

1) On raisonne par contradiction. Supponsons qu'il existe $R_1, R_2 \in \text{Read}_i$ tels que $s(R_1) \neq s(R_2)$. Par hypothèse les lectures sont totalement ordonnées. Supposons que $R_1 \prec_{\text{prec}} R_2$.

- Supposons que $\mu(R_1) = \mu(R_2)$, c'est à dire que dans le programme P, R_1 et R_2 lisent le même emplacement mémoire. Puisque $s(R_1) \neq s(R_2)$, il existe une dépendance anti $d(s(R_1)) \rightarrow s(R_2)$. Donc $d(s(R_1)) \prec_\theta s(R_2)$; Or $R_1 \prec_\theta d(s(R_1)$ (dernière lecture); et $s(R_2) \prec_\theta R_2$ (définition source). Donc: $R_1 \prec_\theta d(s(R_1)) \prec_\theta R_2$, et donc $R_1 \prec_{\text{prec}} d(s(R_1)) \prec_{\text{prec}} R_2$ (ordre des lectures) et donc $\theta_{\text{obs}}(R_1) \ll \theta_{\text{obs}}(d(s(R_1))) \ll \theta_{\text{obs}}(R_2)$ (ordre observé). Comme $d(s(R_1))$ libère e, R_2 doit attendre une nouvelle écriture avant d'être exécutée. Donc $R_2 \notin \text{Read}_i$. Contradiction.

- Supposons que $\mu(R_1) \neq \mu(R_2)$, c'est à dire que R_1 et R_2 ne lisent pas le même emplacement mémoire dans le programme original P. Comme $\sigma(R_1) = \sigma(R_2)$, $\mu(R_1) \not\prec_\theta \mu(R_2)$. Et comme $R_1 \prec_{\text{prec}} R_2$, $R_1 \prec_\theta R_2$. Ces deux éléments impliquent que l'intervalle de vie de $\mu(R_1)$ précède celui de $\mu(R_2)$ dans l'ordre induit par θ. En d'autres termes: $s(R_1) \prec_\theta d(s(R_1)) \prec_\theta s(R_2) \prec_\theta R_2$. D'où l'on tire la même contradiction que précédement: comme R_2 suit $d(s(R_1))$ (ordre des lectures), il existe une écriture entre R_1 et R_2, et donc $R_2 \notin \text{Read}_i$. Contradiction.

Par abus de langage, on parlera de source d'un Read_i. On écrira aussi $X \prec_{\text{prec}} \text{Read}_i$ pour signifier que X est exécuté avant chaque opération de Read_i.

2) Il reste à prouver que la source de chaque Read_i est bien W_i. Par hypothèse, chaque W_i est la source d'un Read_j. Supposons que l'ordre d'exécution dans P soit: $W_1 \prec_\theta \text{Read}_1 \prec_\theta W_2 \prec_\theta \text{Read}_2 \prec_\theta \ldots \prec_\theta W_k \prec_\theta \text{Read}_k$. On a $\theta_{\text{obs}}(W_1) \ll \ldots \ll \theta_{\text{obs}}(W_k)$ et $\theta_{\text{obs}}(\text{Read}_1) \ll \ldots \ll \theta_{\text{obs}}(\text{Read}_k)$.

On observe alors l'ordonnancement suivant: $\theta_{\text{obs}}(W_1) \ll \theta_{\text{obs}}(\text{Read}_1)$ (écriture W_2 bloquante). Après exécution de $d(s(W_1))$, l'emplacement e est à nouveau libre pour une écriture. On enchaine alors sur W_2 (ordre des écritures), et ainsi de suite, de proche en proche. Ainsi, la source de chaque Read_i est bien W_i. □

Il reste maintenant à montrer que les DPN ne produisent jamais d'interblocage.

Théorème 2 (Terminaison) *La compilation de P par le schéma $(\Pi, \sigma, \prec_{prec})$ décrit ci-dessus produit un DPN R exempt d'interblocage.*

Preuve. Supposons obtenu un DPN qui mène à un interblocage. Sans perte de généralité, supposons pour simplifier que le cycle qui présente un interblocage est de taille 2: $P_1 \to P_2 \to P_1$. Appelons C_1 le canal qui relie le processus P_1 au processus P_2, et C_2 le canal qui relie le processus P_2 au processus P_1. Il existe donc $W_1 \in CW(C_1), R_2 \in \mathcal{R}(C_1)$ et $W_2 \in \mathcal{W}(C_2), R_1 \in \mathcal{R}(C_2)$ tels que: $\sigma(W_1) = \sigma(R_2)$, $\sigma(W_2) = \sigma(R_1)$, et: $\theta_{\text{obs}}(W_2) \ll \theta_{\text{obs}}(R_2)$, $\theta_{\text{obs}}(W_1) \ll \theta_{\text{obs}}(R_1)$, et: $s(R_2) = W_1$, $s(R_1) = W_2$.

On a: $W_1 \prec_\theta R_2$ (dépendance de flot), Puisque W_1 est en attente de R_1, il existe une précédence $R_1 \prec_{\text{prec}} W_1$, et donc $R_1 \prec_\theta W_1$ (ordre total sur les processus). De même $W_2 \prec_\theta R_1$ et $R_1 \prec_\theta W_1$ Par transitivité $W_1 \prec_\theta W_1$ c'est à dire $\theta(W_1) < \theta(W_1)$. Contradiction. □

4 Compilation

Cette section détaille étape par étape le fonctionnement d'un compilateur de DPN. La sous-section 4.1 explique comment traduire directement un programme à contrôle statique en DPN, sans optimisation. Les sous-sections suivantes montrent comment transformer le DPN obtenu pour gérer optimalement les entrées/sorties (sous-section 4.2) et pour ajouter davantage de parallélisme (sous-section 4.3).

4.1 Compilation simple

Un DPN simple comporte trois types de processus:

- Des *processus* LD, qui lisent les données entrantes dans la mémoire centrale.
- Des *processus* ST, qui écrivent le résultat final du calcul en mémoire centrale.

- Des *processus* C, en charge du calcul à proprement parler.

On considère un programme P à traduire en DPN et un ordonnancement θ valide.

Afin de construire facilement les processus LD et ST, on ajoute au programme P, pour chaque tableau a lu, la famille d'opérations suivante:

$$\vec{i} \in \Omega : a[\vec{i}] = 0; //(\text{LD}_a, i)$$

Pour chaque tableau écrit b, on ajoute au programme la famille d'opérations suivante, où d est une variable scalaire quelconque:

$$\vec{i} \in \Omega : d = b[\vec{i}]; //(\text{ST}_b, i)$$

On convient que les opérations LD_a sont exécutées tout au début de P, et que les opérations de ST_b sont exécutées tout à la fin de P. Ainsi $(\text{LD}_a, \vec{i_1}) \prec_{\text{prec}} (S, \vec{i_2}) \prec_{\text{prec}} (\text{ST}_b, \vec{i_3})$ pour chaque opération originale $(S, \vec{i_2})$ de P. Cette propriété peut être spécifiée avec un ordonnancement multidimensionnel.

On créé un réseau DPN en plaçant chaque affectation sur un processus différent, $\mathcal{P} = \{S_1, \ldots, S_n\}$ (affectations de P), et $\Pi(W) = \Pi(R_1) = \ldots = \Pi(R_n) = S$. En particulier, on créé un processus LD_a par tableau a lu, et un processus ST_b par tableau b écrit. Nous verrons que la connection de ces processus aux processus de calcul s'obtient naturellement avec la fonction source s.

Chaque processus est muni des éléments suivants:

- Un *port d'entrée* par référence lue par l'affectation implementée par le processus. Le port d'entrée est un *multiplexeur*, chargé de récupérer la valeur de la référence dans le bon canal.
- Un *port de sortie*, qui émet la valeur calculé par le processus. Le port de sortie est un *démultiplexeur* qui écrit la valeur calculée par le processus dans le bon canal.
- Un prédicat de *dernière lecture* dans chaque port d'entrée. Le prédicat vaut vrai si la lecture courante du port d'entrée est la dernière lecture d'une source (un $d(W)$).

Pour constuire un DPN, il suffit de calculer ces éléments, et d'allouer chaque canal (fonction σ).

Multiplexage, démultiplexage

On calcule la fonction source pour chaque lecture. Pour une opération $(S, \vec{i}) : i \in D : W = f(R_1, \ldots, R_n)$, la source de R_k se présente sous la forme:

$$s((S, \vec{i}) : R_k) = \begin{cases} \vec{i} \in D_1 : (T_1, u_1(\vec{i})) \\ \ldots \\ \vec{i} \in D_\ell : (T_\ell, u_\ell(\vec{i})) \end{cases}$$

Les D_k sont des polyèdres convexes fermés, les u_k sont des fonctions affines et les T_k sont des affectations du programme.

Pour chaque couple d'affectations (S, T) telles que S écrit un tableau lu par la référence R_k de T, on créé un canal C_{S,T,R_k}. Si D est le domaine d'itération de S, C_{S,T,R_k} est un tableau de dimension $\dim D$. Ces canaux sont ensuite connectés aux processus *via* les ports d'entrée et de sortie en procédant de la façon suivante.

Pour chaque affectation $S \in \{S_1, \ldots, S_n\}$, pour chaque lecture R_k de S, pour chaque clause $i \in D_k : (T_k, u_k(\vec{i}))$ de la source $s((S, \vec{i}) : R_k)$ (voir ci-dessus), on effectue les opérations suivantes:

- On ajoute au port d'entrée correspondant à R_k une connection au canal C_{T_k,S,R_k}, et on ajoute la clause de multiplexage $\vec{i} \in D_k : C_{T_k,S,R_k}[u_k(\vec{i})]$ qui commande la lecture de la case $u_k(\vec{i})$ du canal C_{T_k,S,R_k} quand l'iteration courante vérifie $\vec{i} \in D_k$.
- On ajoute au port de sortie de T_k une connection vers le canal C_{T_k,S,R_k}, et on ajoute la clause de multiplexage $\vec{i} \in u_k(D_k) : C_{T_k,S,R_k}[\vec{i}]$ qui commande l'écriture de la valeur calculée dans la case $\vec{i}$ (vecteur d'itération courant) du canal C_{T_k,S,R_k} quand l'itération courante vérifie $\vec{i} \in u_k(D_k)$.

Par construction, chaque canal C_{T_k,S,R_k} possède bien une unique source (T_k) et une unique destination (S). Ensuite, les processus LD et ST sont connectés correctement au DPN. En effet, lorsque T_k est un processus LD_a, il s'agit d'une valeur d'entrée du processus, et l'algorithme créée naturellement le multiplexage pour récupérer la bonne valeur dans le canal écrit par LD_a. De plus, le multiplexage d'un processus de sortie ST_b récupère dans ses canaux les valeurs finales pour chaque case de b.

Exemple (suite). On obtient le DPN donné en annexe 1. Les processus sont représentés par des boîtes jaunes. Chaque processus possède plusieurs ports d'entrée (flèches entrantes). Chaque port d'entrée provient d'un multiplexeur (boîte pointue blanche avec le label In_ mux), sur lequel figure le numéro du port d'entrée (sous-label read=#k). Le multiplexeur lit une donnée parmis plusieurs canaux. De même, chaque processus possède un port de sortie (flèche sortante), qui transmet le résultat du calcul courant vers un démultiplexeur (boite blanche pointue renversée avec le label Out_Demux). Le démultiplexeur écrit le résultat dans un ou plusieurs canaux (flèches sortantes).

Le processus 3 (en magenta) calcule l'affectation S, et possède trois ports d'entrée, avec les valeurs de $a[i-1]$ (read 0), $a[i]$ (read 1), et $a[i+1]$ (read 2). Pour la lecture $a[i-1]$, la fonction source $s((S,t,i) : a[i-1])$ indique que quand $i \geq 2 \wedge t = 0$, la source est $(\text{LD}_a, i-1)$, d'où la production d'un buffer en provenance de LD_a (buffer0, en magenta). Le démultiplexage correspondant (Out_demux0, option 0, en provenance du process 0) est donné par $[(t,i) \mapsto i-1](i \geq 2 \wedge t = 0) = i \geq 1$. En procédant de façon analogue pour chaque source de chaque lecture de chaque affectation, on obtient le DPN ainsi donné. Dans la suite, le processus qui exécute l'affectation S sera dénomé "processus S" et le processus qui exécute l'affectation T sera dénommé "processus T". □

Allocation des canaux

Il faut maintenant calculer une fonction d'allocation σ pour chaque canal. On forme donc, pour chaque canal C_{T_k,S,R_k}, un programme P_{T_k,S,R_k} et un ordonnancement θ_{T_k,S,R_k} qui résument le comportement du processus producteur T_k et du processus consommateur S dans C_{T_k,S,R_k}. En somme, P_{T_k,S,R_k} comporte toutes les opérations qui lisent et écrivent C_{T_k,S,R_k}:

$$\begin{array}{ll} \vec{i} \in u_k(D_k) : C_{T_k,S,R_k}[\vec{i}] = d & //S_1 \\ \vec{i} \in D_k : d = C_{T_k,S,R_k}[u_k(\vec{i})]; & //S_2 \end{array}$$

Remarquons qu'un multiplexeur peut avoir plusieurs clauses qui lisent C_{T_k,S,R_k}, correspondant à autant de clauses de démultiplexage qui écrivent C_{T_k,S,R_k}. En conséquence, le programme peut être plus complexe que sur l'illustration.

$\theta_{T_k,S,R_k}(S_1,\vec{i})$ est simplement une restriction de l'ordonnancement original du programme θ' à P_{T_k,S,R_k}: $\theta_{T_k,S,R_k}(S_1,\vec{i}) = \theta'(T_k,\vec{i})$ et $\theta_{T_k,S,R_k}(S_2,\vec{i}) = \theta'(S,\vec{i})$.

On calcule une *allocation* de C_{T_k,S,R_k}, σ_{T_k,S,R_k}, qui est θ_{T_k,S,R_k}-compatible dans P_{T_k,S,R_k} en appliquant un algorithme de repliement mémoire [1]. σ_{T_k,S,R_k} est alors utilisée comme fonction d'adressage du canal. Toute requête vers $C_{T_k,S,R_k}[\vec{i}]$ accède en réalité $C_{T_k,S,R_k}[\sigma_{T_k,S,R_k}(\vec{i})]$. L'allocation se présente sous la forme d'une fonction affine modulaire $A\vec{i} \mod \vec{b}$. Les coordonnées de $\vec{b}$ correspondent aux tailles des différentes dimensions du canal C_{T_k,S,R_k}, qui peut maintenant être effectivement construit.

Exemple (suite). Pour allouer le canal $C_{S,T,b[i]}$ entre S et T, on forme le programme $P_{S,T,b[i]}$ (ici, u_k est l'identité):

$$\begin{array}{ll} 0 \leq t \leq K-2 \wedge 1 \leq i \leq N-2: & C_{S,T,b[i]}[t,i] = d; \\ 0 \leq t \leq K-2 \wedge 1 \leq i \leq N-2: & d = C_{S,T,b[i]}[t,i]; \end{array}$$

avec la restruiction de θ à S et T: $\theta(S,t,i) = (t+i,t,0)$, $\theta(T,t,i) = (t+i,t,1)$. L'outil Bee retourne alors la fonction d'allocation $\sigma(t,i) = (t \mod 1, i \mod 1)$. En d'autre termes, $C_{S,T,b[i]}$ est un canal à 2 dimensions de taille 1×1, qui est dorénavant accédé avec σ. Il peut être implémenté avec un simple registre. □

Synchronisation des canaux

Dans un DPN, les canaux sont bloquants en lecture et en écriture. Une case ne peut pas être lue avant d'avoir été écrite (par sa source), et une case ne pas être réécrite avant que sa valeur soit lue pour la dernière fois. Il faut synchroniser les écritures et les lectures dans un canal pour forcer cette propriété.

On construit une *unité de synchronisation* $\mathbb{S}_c$ pour chaque canal c qui relie le port de sortie d'un processus producteur P au port d'entrée R_k d'un processus consommateur C. Chaque itération W du producteur et chaque itération R du consommateur sont préalablement soumises à l'unité de synchronisation, qui peut décider ou non d'autoriser leur exécution. Pour forcer les dépendances de flot, l'unité de synchronisation décide d'arréter l'exécution du consommateur si la donnée qu'il veut lire n'est pas encore produite. Et pour forcer les dépendances anti-, l'unité de synchronisation décide d'arréter l'exécution du producteur s'il veut écraser une donnée qui n'a pas encore été lue pour la dernière fois. Notons qu'il n'est pas nécessaire de forcer les dépendances de sortie, toutes les écritures étant exécutées par le producteur dans l'ordre séquentiel de l'ordonnancement. Ces deux évenements se traduisent de la façon suivante, où $X \preceq_\theta Y$ signifie que $\theta(X) \lessdot\!\!\!\equiv \theta(Y)$:

$$\mathbb{S}_c(W,R) = \begin{cases} \text{Freeze(C)} & \text{si } W \preceq_\theta s(R) \\ \text{Freeze(P)} & \text{si } \{R' \in \Omega(P,I),\ R \prec_\theta R' \prec_\theta W \wedge \sigma_c(R') = \sigma_c(W)\} \neq \emptyset \end{cases}$$

Le signal Freeze(C) interrompt l'exécution du processus consommateur, et Freeze(P) celle du producteur. Un signal *freeze* existe aussi longtemps que sa condition est vraie. Lorsque sa condition redevient fausse, il cesse d'être émis, et le processus reprend son exécution. La première condition signifie que l'exécution du consommateur est interrompue lorsque la donnée à lire n'est pas encore écrite. Autrement dit, lorsque le producteur exécute une ecriture W qui précéde l'écriture de la donnée lue par R, $s(R)$. La deuxième condition signifie que l'exécution du producteur est interrompue quand il est sur le point d'écraser une donnée qui n'a pas fini d'être lue. Autrement dit, lorsque le producteur tente d'écrire une case $\sigma_c(W)$ qui doit encore être lue plus tard par R'.

La fonction source $s(R)$ est déjà calculée par le multiplexeur du processus C qui lit le canal c. La condition Freeze(P) peut être simplifiée de manière conservative en $R \prec_\theta W$, comme le montre le théorème suivant:

Théorème 3 (Freeze(P) simplifié) *Si Freeze(P) est vrai, on a nécessairement $R \prec_\theta W$.*

Preuve. Si Freeze(P) est vrai, il existe R' avec $R \prec_\theta R' \prec_\theta W$. On obtient le résultat par transitivité de $\prec_\theta$. □

Ce théorème signifie que chaque fois qu'il *faut* bloquer le producteur, $R \prec_\theta W$ devient vrai. Par contre, $R \prec_\theta W$ peut continuer à être vrai plusieurs itérations après que la case à écrire ait été libérée par le consommateur, bloquant inutilement le producteur. En particulier, quand $d(R') \prec_\theta R \prec_\theta W$, le producteur devra encore attendre que R soit exécuté. Au final, le producteur respectera exactement l'ordonnancement θ spécifié par l'utilisateur, sans "faire de zèle". Il est donc de la responsabilité de l'utilisateur de fournir l'ordonnancement le plus parallèle possible.

Le théorème suivant montre la correction des unités de synchronisation.

Théorème 4 (Synchronisation correcte) *L'unité de synchronisation $\mathbb{S}_c$ respecte les dépendances de données et ne produit jamais d'interblocage.*

Preuve. Tout d'abord, on montre qu'une unité de synchronisation force le respect les dépendances (partie **1)**). Cette propriété garantit la correction du calcul réalisé par le réseau de processus. Il suffit de vérifier que les dépendances de flot et les dépendances anti- sont respectées. Ensuite, on montre que les unités de synchronisation ne peuvent jamais bloquer le circuit, ce qui garanti la terminaison du calcul (partie **2)**).

1) Respect des dépendances. Considérons une dépendance de flot $W \rightarrow_{\text{flot}} R$ du processus producteur P vers le processus consommateur C. On a $W \preceq_\theta s(R) \prec_\theta R$ (définition de s(.)). Donc Freeze(C) est vrai, ce qui bloque C et force à exécuter W avant R, la dépendance est donc respectée. Considérons maintenant une dépendance anti- $R \rightarrow_{\text{anti}} W$ de C vers P. Par définition d'une dépendance, $R \prec_{\text{seq}} W$. Comme θ est correct, $R \prec_\theta W$. Donc Freeze(P) est vrai, ce qui bloque P, et force à exécuter R avant W. A nouveau, la dépendance est respectée.

2) Absence d'interblocage. Montrons à présent que les unités de synchronisation ne produisent jamais d'interblocage. Il suffit de prouver qu'il n'existe aucune exécution qui bloque à la fois P et C. Supposons qu'il existe des itérations W et R telles que $\mathbb{S}_c(W, R)$ vaut à la fois Freeze(C) et Freeze(P). On a $W \preceq_\theta s(R)$ (Freeze C) et $R \prec_\theta W$ (Freeze(P)). Comme $s(R) \prec_\theta R$, on a $W \preceq_\theta s(R) \prec_\theta R \prec_\theta W$. Par transitivité on obtient $W \prec_\theta W$. Contradiction. □

Exemple (suite). Considérons à nouveau le canal $c = C_{S,T,b[i]}$ entre les processus S (producteur) et T (consommateur). On a $s(T, t_C, i_C) = (S, t_P, i_P)$, où (t_C, i_C) est l'itération courante du processus consommateur (T), et (t_P, i_P) celle du processus producteur (S). On obtient les prédicats suivants:

$$\begin{aligned}
\text{Freeze}(C) &= W \preceq_\theta s(R) = (t_P + i_P, t_P, 0) \overset{\ll}{=} (t_C + i_C, t_C, 0) \\
&= (t_P + i_P < t_C + i_C) \vee (t_P + i_P = t_C + i_C \wedge t_P < t_C) \\
&\quad \vee (t_P + i_P = t_C + i_C \wedge t_P = t_C) \\
\text{Freeze}(P) &= R \prec_\theta W = (t_C + i_C, t_C, 1) \ll (t_P + i_P, t_P, 0) \\
&= (t_C + i_C < t_P + i_P) \vee (t_C + i_C = t_P + i_P \wedge t_C < t_P)
\end{aligned}$$

Par nature, l'ordre lexicographique ($\ll$) produit des expressions très redondantes, ce qui permet de produire un circuit de test très simple. □

Génération des processus

Enfin, il reste à générer l'automate de contrôle et le chemin de données pour chaque processus. Par construction, chaque processus exécute une affectation S du programme sur tout son domaine d'itération D_S. Ensuite, on connait l'ordonancement θ du programme, et en particulier sa restriction à l'affectation S, θ_S. Il suffit de donner D_S et θ_S à un générateur de code polyédrique [4] pour obtenir un automate de contrôle directement exploitable. Le chemin de données de l'opérateur exécuté par le processus peut etre généré automatiquement à partir de son expression arithmétique [5]. Le chemin de données est généralement pipeliné et il faut ordonnancer les processus pour éviter de bloquer le pipeline [3].

Exemple (suite). Pour le processus S, on obtient $First() = (0, 1)$ et la fonction Next() suivante. Pour simplifier, on suppose que le domaine d'itération est un carré: $K = N - 2$.

$$Next(t,i) = \begin{cases} i \geq 2 \wedge t \leq K-2: & (t+1, i-1) \\ i = 1 \wedge t+i < K: & (0, t+i+1) \\ i = 1 \wedge t+i = K: & (1, t+i) \\ t = K-2 \wedge t+i > K: & (i+K-N+1, N-2) \end{cases}$$

La première clause décrit le segment $t + i =$ constante en remontant. Une fois "en haut", il faut redescendre et se placer au début du segment $t + i =$ constante $+ 1$. En dessous (strictement) de la diagonale du carré, on redescend à $t = 0$ (clause 2). Sur la diagonale, on descend à $(1, t + i)$. La dernière clause s'applique quand on atteint le segment supérieur du carré ($t = K - 2$). On vérifie que pour chacune de ces trois clauses, la quantité $t + i$ est incrémentée. □

4.2 Optimisation des entrées/sorties

Dans des travaux précédents [2], nous avons montré comment les entrées/sorties peuvent être générés pour utiliser optimalement la bande passante et réutiliser optimalement les données chargées, tout en gardant une mémoire locale de taille raisonnable. Nous proposons ici une généralisation, sans perte d'optimalité, de ces travaux à un réseau de processus DPN. Dans un premier temps, les processus sont divisés en groupes (sous-section 4.2.2). Ensuite, pour chaque groupe, les entrées/sorties sont optimisées en produisant les processus LD et ST et le multiplexage approprié pour pipeliner optimalement les communications (sous-section 4.2.3). La sous-section suivante commence par rappeller la notion de tuilage.

4.2.1 Tuiles, bandes, plans

Un *tuilage* du domaine d'itération D_S d'une affectation S est un découpage de D_S à l'aide de d hyperplans de tuilage linéairement indépendants, décrits par leurs vecteurs normaux $\mathcal{H}(S) = \{\vec{H_1}, \ldots, \vec{H_d}\}$. Intuitivement, on tranche le domaine d'iteration D_S en translatant les hyperplans à intervalles réguliers de longueur T. Pour chaque dimension $k = 1, d$, on ajoute les contraintes suivantes à D_S: $T.I_k \leq \vec{H_k} \cdot \vec{i} < T.(I_k + 1)$, où les I_k sont de nouvelles dimensions de D_S. Pour chaque valeur possible de $(I_1, \ldots, I_d)$, on obtient un hypercube appelé *tuile*, qui a vacation à être exécuté de façon atomique, on exécute d'abord toutes les opérations d'une tuile, ensuite toutes les opérations d'une autres, etc. On le spécifie pour cela avec un ordonnancement θ_T:

$$\theta_T(S, I_1, \ldots, I_d, \vec{i}) = (I_1, \ldots, I_d, \vec{H_1} \cdot \vec{i}, \ldots, \vec{H_d} \cdot \vec{i})$$

Dans la suite, on écrira $\vec{i}$ pour $I_1, \ldots, I_d, \vec{i}$. Bien entendu, tous les tuilages ne sont pas valides. Pour qu'un tuilage soit *valide*, il faut choisir les hyperplans de sorte qu'aucune dépendance ne

"rentre", $\vec{H_k} \cdot \vec{\delta} \geq 0$ pour tout vecteur de dépendance δ (reliant la source et la destination de la dépendance). Autrement, on obtiendrait une dépendance cyclique entre deux tuiles.

En général, un ordonnancement tuilé peut être plus compliqué, et décrire une alternance de boucles et de boucles tuilées. Des dimensions constantes peuvent également être ajoutées pour exprimer un séquence entre deux nids de boucle. En toute généralité, il existe donc des rangs $n_1^S < \ldots < n_d^S < m_1^S < \ldots m_d^s$ dans cet ordre, tels que pour tout $k = 1, d$: $\theta_T(S, \vec{i})[n_k^S] = I_k$ et $\theta_T(S, \vec{i})[m_k^S] = \vec{H_k} \cdot \vec{i}$. Deux affectations S et T ont des ordonnancements *compatibles* quand $\#\mathcal{H}(S) = \#\mathcal{H}(T)$ et $n_k^S = n_k^T, m_k^S = m_k^T, \forall k = 1, d$. Si $\vec{x} \in \mathbb{N}^p$, $\vec{y} \in \mathbb{N}^q$ et $\ell \leq \min(p, q)$, on défini l'ordre lexicographique au rang ℓ par:

$$\vec{x} \ll^{\ell} \vec{y} \equiv x[0] = y[0] \wedge \ldots \wedge x[\ell - 1] = y[\ell - 1] \wedge x[\ell] < y[\ell]$$

On défini également $\ll^{\leq \ell} = \ll^0 \cup \ldots \cup \ll^{\ell}$ et enfin $\ll^{\geq \ell} = \ll - \ll^{\leq \ell - 1}$.

Une *bande* de S est un sous ensemble de tuiles obtenu en fixant toutes les dimensions de l'ordonnancement de S jusqu'au dernier compte-tour I_d, et en faisant itérer les dimensions à partir de I_d. Formellement, $(S, \vec{i})$ et $(S, \vec{j})$ appartiennent à la même bande de S ssi $\theta_T(S, \vec{i}) \ll^{\geq n_d^S} \theta_T(S, \vec{j})$ ou $\theta_T(S, \vec{j}) \ll^{\geq n_d^S} \theta_T(S, \vec{i})$. Il s'agit d'une relation d'équivalence, et la classe de $(S, \vec{i})$ est noté $\textsc{Band}_{\theta_T}(S, \vec{i})$.

Un *plan* de S est un sous-ensemble de tuiles obtenu en itérant à partir de l'avant dernier compte-tour I_{d-1}. De façon analogue, on défini une relation d'équivalence avec $\ll^{\geq n_{d-1}^S}$, et on note $\textsc{Plan}_{\theta_T}(S, \vec{i})$ la classe de $(S, \vec{i})$.

Exemple (suite). On tuile les affectations S et$T$ avec les hyperplans $\tau_1 = (1, 0)$ ("t = constante") et $\tau_2 = (1, 1)$ ("$t + i$ = constante") et : $\mathcal{H}(S) = \mathcal{H}(T)\{\tau_1, \tau_2\}$. On appelle I_1 et I_2 les compte tours des boucles tuilées, et on ajoute aux domaines d'itération de S et T les contraintes suivantes: $2I_1 \leq t < 2(I_1 + 1) \wedge 2I_2 \leq t + i < 2(I_2 + 1)$, la taille des tuiles est 2 dans la direction τ_1, et 2 dans la direction τ_2. Les domaines obtenus sont représentés à la figure 2. Les nouveaux ordonnancements de S et T sont $\theta(S, I_1, I_2, t, i) = (1, I_1, I_2, t, i, 0)$ et $\theta(T, I_1, I_2, t, i) = (1, I_1, I_2, t, i, 1)$. Dans chaque tuile, on exécute dans l'ordre spécifié sur le dessin (flèches vertes). □

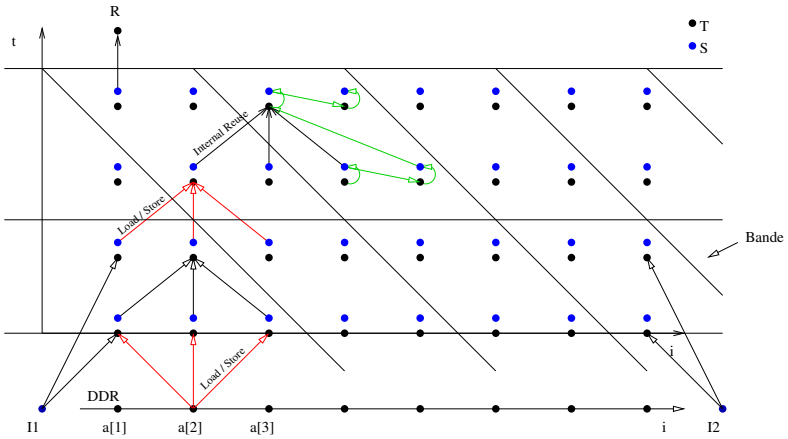


Figure 2: Jacobi 1D tuilé

4.2.2 Groupe de tuiles

L'approche décrite dans [2] charge et stocke les données des tuiles d'une bande en suivant un schéma d'exécution pipeliné, et fonctionne pour un programme réduit à une seule affectation tuilée. En général, un programme comporte plusieurs affectations, et plusieurs affectations peuvent être tuilées. Les exécutions de deux affectations tuilées peuvent s'entrelacer, et les faire appartenir "aux mêmes bandes". On dit alors qu'elles appartiennent au même *groupe de tuile*, et on peut leur appliquer notre schéma d'optimisation des entrées/sorties. L'objet de cett section est de définir précisement cette notion.

L'*ensemble préfixe* d'une partie $E = \{x_0, \ldots, x_{k-1}\} \subset \mathbb{N}$ avec $x_0 < \ldots < x_{k-1}$, est:

$$\mathrm{Pref}(E) = \{\emptyset, \{x_0\}, \{x_0, x_1\}, \ldots, \{x_0, \ldots, x_{k-1}\}\}$$

Etant donnés $E, F \subset \mathbb{N}$, on dit que E est un *préfixe* de F si F commence par les entiers de E, c'est à dire $E \in \mathrm{Pref}(F)$, on note alors $E \sqsubseteq F$. On pose $\sqsupset\!\sqsubset = \sqsubseteq \cup \sqsubseteq^{-1}$, $E \sqsupset\!\sqsubset F$ signifie que l'un des deux ensembles E et F "commence" l'autre. Enfin, pour $\vec{x} = (x_0, \ldots, x_{k-1}) \in \mathbb{N}^k$ et $i = 1, k$, on écrit $\pi_i(\vec{x}) = x_i$. Par extension, pour $E \subset \mathbb{N}^k$, $\pi_i(E) = \{\pi_i(\vec{x}), \vec{x} \in E\}$. On étend la relation $\sqsupset\!\sqsubset$ aux ensembles de vecteurs d'entiers de taille k en convenant que pour $E, F \subset \mathbb{N}^k$, on a:

$$E \sqsupset\!\sqsubset F \Leftrightarrow \begin{cases} \pi_0(E) \sqsupset\!\sqsubset \pi_0(F) \wedge \\ \pi_1(\{\vec{x} \in E, x_0 = \lambda_0\}) \sqsupset\!\sqsubset \pi_1(\{\vec{x} \in F, x_0 = \lambda_0\}) \wedge \\ \pi_2(\{\vec{x} \in E, x_0 = \lambda_0 \wedge x_1 = \lambda_1\}) \sqsupset\!\sqsubset \pi_2(\{\vec{x} \in F, x_0 = \lambda_0 \wedge x_1 = \lambda_1\}) \wedge \\ \ldots \\ \pi_k(\{\vec{x} \in E, \wedge_{\ell=0}^{k-1} x_\ell = \lambda_\ell\}) \sqsupset\!\sqsubset \pi_k(\{\vec{x} \in F, \wedge_{\ell=0}^{k-1} x_\ell = \lambda_\ell\}) \end{cases}$$
pour tout $(\lambda_0, \ldots, \lambda_{k-1}) \in \mathbb{N}^{k-1}$

Si on voit E et F comme des ensembles d'ordonnancements classés dans l'ordre lexicographique, $E \sqsupset\!\sqsubset F$ signifie qu'à chaque profondeur d'imbrication, les compte-tour de l'un décrivent un ensemble prefixe des compte tour de l'autre. En somme, à chaque profondeur, ils commencent en même temps et font la même chose, et l'un peut s'arrêter avant l'autre. Par exemple, avec $E = \{(0,0),(0,1),(0,2),(1,0)\} et F = \{(0,0),(1,0),(1,1),(2,0)\}$, on a $E \sqsupset\!\sqsubset F$. En effet, sur la première dimension, E itère sur $\{0,1\}$ et F sur $\{0,1,2\}$. Sur la deuxième dimension, pour $\lambda_0 = 0$, F est prefixe; pour $\lambda_0 = 1, 2$ E est préfixe. Par analogie avec l'ordre lexicographique, on note $\sqsupset\!\sqsubset^{\leq \ell}$ l'opérateur défini uniquement par les ℓ premières clauses de la conjonction. $E \sqsupset\!\sqsubset^{\leq \ell} F$ signifie que la propriété est vérifié à chaque profondeur *jusqu'à* ℓ.

En somme la relation $\sqsupset\!\sqsubset^{\leq \ell}$ caractèrise un d'entrelacement, calé sur la première itération à chaque profondeur. C'est exactement l'outil qu'il nous faut pour caractériser l'appartenance de deux affectations au même groupe de tuile:

Définition 1 (Groupe de tuiles) *Soient deux affectations $S, T \in \mathcal{S} = \{S_1, \ldots, S_n\}$. On défini la relation $\sqcup\!\sqcup$ par:*

$$S \sqcup\!\sqcup T \Leftrightarrow \begin{array}{l} \textit{(i) } S \textit{ et } T \textit{ ont des ordonnancements compatibles} \\ \textit{(ii) } \theta_T(S, \vec{i}), \vec{i} \in D_S\} \sqsupset\!\sqsubset^{\leq n_{d-1}^S} \theta_T(T, \vec{i}), \vec{i} \in D_T\} \end{array}$$

Un groupe de tuiles *est un élement de $\mathcal{G}(P, \theta_T) = \mathcal{S}/\sqcup\!\sqcup = \{G_1, \ldots, G_p\}$.*

Le fait que les affectations d'un groupe commencent en même temps, sans déphasage sur toutes les itérations $\leq n_{d-1}^s$ est essentiel pour garantir le respect des dépendances de données entre les stockages et chargements qui seront ajoutés dans la section suivante.

Il faut de plus interdire que les fausses relations $S \not\perp\!\!\!\perp T$, où S et T ont deux bandes qui s'entrelacent, mais qui ne sont pas reconnues équivalentes à cause d'un déphasage. Pour faciliter les synchronisations, on impose que de telles affectations S et T ne partagent aucune bande, d'où la définition suivante:

Définition 2 (DPN-compatibilité) *Un ordonnancement tuilé est DPN-compatible si:*

- *Chaque affectation tuilée appartient à un groupe de tuiles*
- *Si S et T appartiennent à des groupes de tuiles différents, $S \not\perp\!\!\!\perp T$, Alors:*

$$\textsc{Band}_{\theta_T}(S) \cap \textsc{Band}_{\theta_T}(T) = \emptyset$$

Exemple (suite). Les ordonnancements de S et T sont compatibles, ils ont autant de dimensions tuilées, et les dimensions tuilées sont les mêmes. Par ailleurs, l'ensemble des dates de S et de T sont les mêmes (à cela près que les dates de S terminent par 0, et que les dates de T terminent par 1). En particulier, ils partagent donc les mêmes bandes $(1, I_1)$, d'où $S \mathrel{\underline{\underline{\perp\!\!\!\perp}}}^{\leq n_1^S} T$. Ainsi le programme possède un unique groupe de tuiles $\{S, T\}$. □

4.2.3 Production des communications

Cette sous-section décrit étape par étape les transformations à appliquer au DPN simple obtenu en sous-section 4.1 pour optimiser les entrées/sorties. On suppose donné un ordonnancement θ_T DPN-compatible, et une division du programme en groupes de tuiles.

Pour chaque groupe de tuiles G, on génére les éléments suivants:

- *Les processus de chargement* (Ld) et de *stockage* (St) vers une mémoire distante pour chaque affectation du groupe de tuiles. On créé un processus $\textsc{Ld}_{G,a}$ par tableau a lu dans le groupe, et un processus $\textsc{St}_{G,b}$ par tableau b écrit dans le groupe. L'objectif est de suivre le schéma de pipeline décrit dans [2]. Chaque bande tuilée est considérée séparément. Sur une bande tuilée, on exécute de manière pipelinée la boucle suivante: pour chaque tuile de la bande: (i) charger les données lues par la tuile et *produites à l'extérieur de la bande*, (ii) exécuter la tuile, (iii) stocker les données écrites par la tuile et *utilisées à l'extérieur de la bande*.

  Cette partie est décrite dans les paragraphes **a)** et **b)**.

- *Les canaux* dans lesquels les processus Ld verseront les données lues pour la tuile courante, et les canaux dans lesquels les processus St lisent les résultats du calcul de la tuile courante. Ces canaux devront être dimensionnés pour permettre une exécution pipelinée Ld, St/calcul. Comme précédement, on créé un canal C_{a,S,R_k} par tableau a lu dans le groupe, et par affectation S dont la lecture R_k lit a. De plus, on créé un canal $C_{a,S}$ par tableau a écrit dans le groupe, et par affectation S qui l'écrit. Cette multiplication des canaux est essentielle pour garantir le parallélisme. Toutefois seuls les canaux correspondant à une entrée ou une sortie sont gardés.

  Cette partie est décrite dans les paragraphes **c)** et **d)**.

- *Le multiplexage/démultiplexage* nécessaire pour lire les données des processus Ld, et écrire les données vers les buffers des processus St.

  Cette partie est décrite dans le paragraphe **a)**.

- *Les synchronisations entre processus* LD et ST pour permettre l'exécution pipelinée. Cette partie est décrite dans le paragraphe **e)**.
- *Les synchronisations entre groupes de tuile* garantissant l'exécution dans l'ordre $\prec_{\theta_T}$. Cette partie est décrite dans le paragraphe **f)**.

a) Entrées/sorties d'une tuile, multiplexage/démultiplexage vers LD et ST Ce paragraphe explique comment calculer l'ensemble des données à charger et à stocker pour une tuile. Chaque multiplexeur de chaque processus de calcul du groupe de tuile est inspecté, et quand il est établi que parmis ses lectures, certaines proviennent de l'extérieur de la bande courante, le multiplexeur est modifié pour les lire à partir d'un canal provenant d'un processus LD. De façon symétrique, on "patche" les clauses de démultiplexage qui écrivent une donnée lue à l'extérieur de la bande courante pour écrire ces données dans le canal d'un processus ST. Au final, on obtient à la fois l'ensemble des données à charger et à stocker, et le multiplexage/démultiplexage vers les processus LD et ST du groupe de tuile.

Pour déterminer ce qu'il faut charger et sauvegarder pour une tuile, il faut décider quelles sont les données extérieures. Ce sont simplement les données qui proviennent de l'exterieur de la bande courante. En conséquence, on charge les données $\mu(R_k)$ dont la source $s(R_k)$ appartient à une bande différente (flèches rouges sur la figure 2); et coté source, il faudra stocker la donnée. On transforme ainsi une dépendance *via* un canal en dépendance *via* la mémoire distante.
(i) Entrées (loads). Etant donnée une opération $(S,\vec{i})$ effectuant une lecture R_k, il faut charger $\mu(R_k)$ si le prédicat suivant est vrai:

$$\mathbb{P}_{\text{LD}}(R_k) \equiv \begin{array}{l} s(R_k) \notin Cl_{\sqcup\sqcup}(S) \vee \\ (s(R_k) \in Cl_{\sqcup\sqcup}(S) \wedge \theta_T(s(R_k)) \ll^{\leq n^S_{d-1}} \theta_T(S,\vec{i})) \end{array}$$

En effet, une source extérieure appartient soit à un autre groupe de tuiles, soit au même groupe de tuile, mais dans une bande (ou un plan) différent. Supposons qu'on ait obtenu la fonction source suivante pour le tableau a lu par la référence R_k de l'affectation S:

$$s((S,\vec{i}) : R_k) = \left\{ \begin{array}{l} \vec{i} \in D_1 : (T_1, u_1(\vec{i})) \\ \ldots \\ \vec{i} \in D_\ell : (T_\ell, u_\ell(\vec{i})) \end{array} \right.$$

L'ensemble des données à charger pour la tuile $T \subset D_S$ est donnée par:

$$\text{LD}(a, S : R_k, T) = \mu(T \cap \bigcup_{k=0}^{\ell} (T_k \notin Cl_{\sqcup\sqcup}(S) ? D_k : \{\vec{i} \in D_k,\ \theta_T(T_k, u_k(\vec{i})) \ll^{\leq n^S_{d-1}} \theta_T(S,\vec{i})\}))$$

On applique simplement notre prédicat à chaque clause de la fonction source: si la source provient d'un groupe de tuiles différent (avant ?), alors tout doit être chargé (on accumule D_k), sinon (après :), on sélectionne les itérations de D_k dont la source provient d'une autre bande, et on accumule. Une fois l'union effectué, on dispose de toutes les itérations de S où R_k lit une donnée extérieur. Pour savoir ce qu'il faut lire sur une tuile T, il suffit d'intersecter avec T (paramètre). Et pour savoir de quelles données il s'agit, il suffit de prendre l'image de l'ensemble d'itérations obtenu par la fonction d'index μ de R_k. On obtient des vecteurs de la forme $(\vec{T},\vec{e})$ où $\vec{T}$ désigne la tuile sur laquelle effectuer le transfert de données (tuile T dans les formules ci-dessus), et $\vec{e}$ désigne l'index de a à transférer.

Ensuite, l'ensemble d'iterations obtenu est ajouté comme clause au multiplexeur de R_k ($k$ième port d'entrée du processus $\Pi(S)$). Le canal lu est C_{a,S,R_k}, comme R_k lit le tableau a; et la fonction d'accés au canal est la fonction d'index originale $\mu(R_k)$.

Enfin, l'ensemble $\textsc{Ld}(a, S : R_k, T)$ est ajouté au domaine d'itération du processus $\textsc{Ld}_{G,a}$.

(ii) Sorties (stores). Les stockages sont calculent de façon analogue, en remarquant que lorsque la source $s(R_k)$ appartient à une bande différente, il faut en plus faire un store du coté de $s(R_k)$. On reprend donc l'algorithme décrit ci-dessus (avec l'union), et on le transforme de la façon suivante:

Pour chaque clause k de $s((S, \vec{i}) : R_k)$:

- Si $T_k \notin Cl_{\sqcup\sqcup}(S)$ Alors ajouter $\mu(T \cap u_k(D_k))$ à $\textsc{St}(a, T_k)$
- Sinon: Ajouter $\mu(T \cap \{u_k(\vec{i}),\ \vec{i} \in D_k \wedge \theta_T(T_k, u_k(\vec{i})) \ll^{\leq n^S_{d-1}} \theta_T(S, \vec{i})\})$ à $\textsc{St}(a, T_k)$.

En effet, si la source appartient à un autre groupe il faut tout charger (donc tout stocker coté St), sinon, on ne stocke que ce qui correspond à une dépendance sortante, de façon totalement symétrique au calcul du Ld. Ensuite, l'ensemble d'iterations obtenu pour chaque clause k est ajouté comme clause au démultiplexeur de T_k. Le canal écrit est C_{a,T_k}, comme T_k écrit le tableau a; et la fonction d'accés au canal est la fonction d'index originale de l'écriture dans T_k: $\mu(W)$. Enfin, l'ensemble $\textsc{St}(a, Tk)$ est ajouté au domaine d'itération du processus $\textsc{St}_{Cl_{\sqcup\sqcup}(T_k),a}$.

Les domaines d'itérations ainsi calculés s'écrivent comme l'image d'un polyédre par une fonction affine (μ). En toute généralité, ce ne sont plus des polyédres, mais des LBL (linearly bounded lattice).

Exemple (suite). Considérons le processus S et son multiplexeur sélectionnant la valeur de $a[i-1]$. On analyse chaque entrée de multiplexage pour déterminer si elle provient de $\textsc{Ld}_{G,a}$.

- L'entrée qui provient de I_1 n'est pas prise en compte. Elle ne provient pas d'un groupe de tuiles, et son canal est maintenu tel quel.
- L'entrée qui provient de $\textsc{Ld}_{G,a}$ est active lorsque $t = 0 \wedge 2 \leq i \leq N-2$. Les entrées d'un groupe de tuile qui sont des inputs du programme doivent être chargées par $\textsc{Ld}_{G,a}$. On ajoute donc au domaine d'itération de $\textsc{Ld}_{G,a}$ l'ensemble suivant:

$$\begin{aligned}&[(I_1, I_2, t, i) \mapsto i-1]\{(I_1, I_2, t, i), 2 \leq i \leq N-2 \wedge t = 0 \wedge \\ &\qquad\qquad 2I'_1 \leq t < 2(I'_1+1) \wedge 2I'_2 \leq t+i < 2(I'_2+1)\} \\ &\quad = \{i,\ 1 \leq i \leq N-3 \wedge I'_1 = 0 \wedge 2I'_2 \leq i < 2(I'_2+1)\}\end{aligned}$$

  (I'_1, I'_2) est un paramètre qui représente la tuile T. Ainsi, les contraintes avec I'_1 et I'_2 expriment l'intersection avec la tuile T. En fait, chaque point de l'ensemble obtenu est un vecteur (I'_1, I'_2, i). La condition de multiplexage est conservée, par contre on lit la case $i-1$ du canal $C_{a,S,a[i-1]}$.

- L'entrée qui provient de T est active lorsque $t \geq 1 \wedge 2 \leq i \leq N-2$. Elle provient d'une autre bande lorque $t-1 < 2I_1 \wedge t-1 \geq 2I_1$ ($\ll^{n^S_1}$). On ajoute donc au domaine d'itération de $\textsc{Ld}_{G,a}$ l'ensemble

$$\begin{aligned}&[(I_1, I_2, t, i) \mapsto i-1]\{(I_1, I_2, t, i), t-1 < 2I_1 \wedge t \geq 2I_1 \wedge i \geq 2 \wedge t \geq 1 \wedge \\ &\qquad\qquad 2I'_1 \leq t < 2(I'_1+1) \wedge 2I'_2 \leq t+i < 2(I'_2+1)\} \\ &\quad = \{i,\ 1 \leq i \leq N-3 \wedge 2(I'_2 - I'_1) \leq i \leq 2(I'_2 - I'_1)+1\}\end{aligned}$$

On ajoute une clause de multiplexage $M = \{(I_1, I_2, t, i),\ t-1 < 2I_1 \wedge t \geq 2I_1 \wedge i \geq 2 \wedge t \geq 1\}$, qui lit la case $i-1$ du canal $C_{a,S,a[i-1]}$. Il s'agit d'un sous-cas de l'entrée traitée. Elle devra être choisie prioritairement (si elle est vraie, c'est elle qui est choisie).

A ces loads, correspondent des stores coté T. Ici, on lit la case $(t-1, i-1)$ du canal provenant de T, $u_k(I_1, I_2, t, i) = (t-1, i-1)$. Après calcul, la clause de démultiplexage à ajouter à T est: $u_k(M) = \{(I_1, I_2, t, i),\ t = I_1 + 1 \wedge 1 \leq i \leq N-3\}$. On écrit alors la case i du nouveau canal $C_{a,T}$. De façon analogue, on ajoute ensuite au domaine d'itération de $\mathrm{ST}_{G,a}$ l'ensemble des données écrites $\mu(u_k(M) \cap T) = \{(I'_1, I'_2, i),\ 1 \leq i \leq N-3 \wedge 2(I'_2 - I'_1) \leq i < 2(I'_2 - I'_1 + 1)\}$.

Le DPN obtenu est donné en annexe 2. Les nouveaux processus $\mathrm{LD}_{G,a}$ et $\mathrm{ST}_{G,a}$ et les canaux qui viennent d'être ajoutés sont représentés en magenta. L'option 2 de In_mux0 correspond au cas $t = 0 \wedge 2 \leq i \leq N-2$. L'option 4 correspond au cas $t \geq 1 \wedge 2 \leq i \leq N-2$. Les options de démultiplexage correspondantes sont les options 0 et 1 de Out_Demux 7 (en provenance du processus 7, $\mathrm{LD}_{0,a}$). On applique cette transformation sur tous les multiplexeurs de tous les processus du groupe (S et T) pour obtenir le DPN complet. □

Réutilisation des données chargées. Le procédé décrit ci-dessus permet de charger et de stocker uniquement les données produites et utilisées à l'extérieur de la bande de tuiles. De cette façon, les dépendances internes à la bande de tuile (d'une tuile vers une autre) sont résolues par des buffers internes (flèches noires sur la figure 2). On évite ainsi de les stocker une fois la tuile productrice exécutée, pour ensuite les recharger avant d'exécuter la tuile consommatrice. On dit que ces données sont *réutilisées*. Par contre, notre procédé ne permet pas, tel quel, de réutiliser une donnée extérieure lue par plusieurs tuiles de la bande. Chaque tuile devra la recharger, alors même qu'il suffit de la charger une fois pour toutes *avant sa première lecture*, et ensuite la conserver dans un buffer interne, qui sera lu par les autres tuiles. On dit que la donnée est réutilisée (dans le buffer). Notons que ce problème n'affecte pas les stockages, puisqu'une fonction source $s(.)$ extérieure à la bande de tuile sélectionne, par définition, la dernière écriture de la donnée dans la bande de tuile. Ce paragraphe montre une transformation simple du domaine d'itération pour chaque processus $\mathrm{LD}_{G,a}$ pour permettre une réutilisation optimale des données chargées. On peut le voir comme une post-passe optimisante.

Pour simplifier la présentation, on reprend, sans perte de généralité, les notations de l'exemple. Le domaine d'itération de $\mathrm{LD}_{G,a}$ est un ensemble $\mathcal{D}_{\mathrm{LD}_{G,a}}$ de couples (I_1, I_2, i), ce qui signifie que pour la tuile de coordonnées (I_1, I_2), il faut charger la donnée $a[i]$. Si $a[i]$ est chargée par une autre tuile (I'_1, I'_2) , on aura de même $(I'_1, I'_2, i) \in \mathcal{D}_{\mathrm{LD}_{G,a}}$. Comme on l'a vu, on ne veut faire le chargement de $a[i]$ que pour la *première de ces tuiles*, c'est à dire pour la tuile $T(i) = \min_{\prec_\theta}\{(I_1, I_2),\ (I_1, I_2, i) \in \mathcal{D}_{\mathrm{LD}_{G,a}}\}$. On écrit $T(i)$ sous la forme $\min_{\ll}\{(\theta(I_1, I_2, i), I_1, I_2),\ (I_1, I_2, i) \in \mathcal{D}_{\mathrm{LD}_{G,a}}\}$, qui s'exprime à l'aide d'un minimum lexicographique, calculable par un solveur. On obtient un résultat de la forme suivante, où les D_k sont des polyèdres, et les T_k sont des fonctions affines:

$$T(i) = \begin{cases} i \in D_1 : T_1(i) \\ \dots \\ i \in D_n : T_n(i) \end{cases}$$

On remplace alors le domaine d'itération de $\mathrm{LD}_{G,a}$ par le domaine optimisé suivant:

$$\mathcal{D}^{\mathrm{opt}}_{\mathrm{LD}_{G,a}} = \bigcup_{k=1}^{n} \{(I_1, I_2, i),\ i \in D_k \wedge (I_1, I_2) = T_k(i)\}$$

Exemple (suite). Considérons à nouveau le processus S et, cette fois ci, ses trois multiplexeurs, sélectionnant respectivement les valeurs de $a[i-1]$, $a[i]$ et $a[i+1]$. En appliquant le procédé décrit dans le paragraphe précédent, on obtient le domaine:

$$\mathcal{D}_{\mathrm{LD}_{G,a}} = \{(I_1, I_2, i),\ 2(I_1 - I_2) \leq i \leq 2(I_1 - I_2) + 3\}$$

Ce domaine décrit des chargements redondants. En effet, la tuile (I_1, I_2) charge les données suivantes:

$$\{a[2(I_1 - I_2)], a[2(I_1 - I_2) + 1], a[2(I_1 - I_2) + 2], a[2(I_1 - I_2) + 3]\}$$

Et la tuile suivante $(I_1, I_2 + 1)$ charge les données: $\{a[2(I_1 + 1 - I_2)], a[2(I_1 + 1 - I_2) + 1], a[2(I_1 + 1 - I_2) + 2], a[2(I_1 + 1 - I_2) + 3]\}$, c'est à dire:

$$\{a[2(I_1 - I_2) + 2], a[2(I_1 - I_2) + 3], a[2(I_1 - I_2) + 4], a[2(I_1 - I_2) + 5]\}$$

Ainsi les données $a[2(I_1 - I_2) + 2]$ et $a[2(I_1 - I_2) + 3]$ sont chargées par 2 tuiles consécutives. En appliquant le procédé décrit dans ce paragraphe, on obtient le domaine:

$$\begin{aligned} \mathcal{D}^{\mathrm{opt}}_{\mathrm{LD}_{G,a}} = \quad & \{(I_1, I_2, i),\ 2(I_1 - I_2) \leq i \leq 2(I_1 - I_2) + 3 \wedge I_2 = I_1\} \cup \\ & \{(I_1, I_2, i),\ 2(I_1 - I_2) + 2 \leq i \leq 2(I_1 - I_2) + 3 \wedge I_2 > I_1\} \end{aligned}$$

Le premier ensemble décrit les données à charger pour la première tuile de la bande ($I_2 = I_1$). Les données chargées sont $a[0], a[1], a[2], a[3]$. Le deuxième ensemble décrit les données à charger pour les tuiles suivantes ($I_2 > I_1$). Seules les deux données suivantes sont chargées. Par exemple $a[4], a[5]$ pour la deuxième tuile, puis $a[6], a[7]$ pour la troisième tuile, etc. □

b) Production des processus LD et ST Une fois déterminées les données à lire et à écrire pour un groupe de tuile G, et le multiplexage/démultiplexage corrigé pour lire et écrire dans les canaux des processus $\mathrm{LD}_{G,a}$ et $\mathrm{ST}_{G,a}$ correspondants, il faut compiler ces processus LD et ST, c'est à dire produire le chemin de données pour réaliser les requêtes d'entrées/sortie avec la mémoire externe, et produire l'automate de contrôle.

Sur les mémoires modernes, le transfert se fait en mode *burst* par blocs de taille fixe Λ (*chunks*). Pour pouvoir itérer sur les chunks à charger/stocker, on ajoute aux domaines d'itération une dimension c qui indique à quel numéro de chunk la donnée $a[\vec{e}]$ appartient, et une dimension o qui indique le décalage (*offset*) de la donnée dans le chunk. On suppose que les tableaux sont organisés par lignes et que leurs dimensions sont des constantes déclarées par l'utilisateur. Ainsi, l'adresse physique de $a[\vec{e}]$, notée $@a[\vec{e}]$ est toujours une forme affine en $\vec{e}$.

On étend donc le domaine d'itération $\mathcal{D}$ de $\mathrm{LD}_{G,a}$ en:

$$\hat{\mathcal{D}} = \{(\vec{T}, c, o, \vec{e}),\ \Lambda c \leq @a[\vec{e}] < \Lambda(c+1) \wedge o = @a[\vec{e}] - \Lambda c \wedge (\vec{T}, \vec{e}) \in \mathcal{D}\}$$

Le domaine d'itération du processus s'obtient en projetant $\hat{\mathcal{D}}$ sur $\vec{T}$ et c, autrement dit en éliminant les variables o et $\vec{e}$ de $\hat{\mathcal{D}}$:

$$\mathcal{D}_{\mathrm{LD},G,a} = \{(\vec{T}, c),\ \exists o, \vec{e}\ \text{t.q.}\ (\vec{T}, c, o, \vec{e}) \in \hat{\mathcal{D}}\}$$

Pour produire le contrôle, il reste à ordonnancer par tuiles croissantes, puis par chunks croissants (dans cet ordre) avec l'ordonnancement $\theta_{\mathrm{LD},G,a}(\vec{T}, c) = (\vec{T}, c)$. Comme pour le DPN simple, l'automate de contrôle est obtenu en donnant $\mathcal{D}_{\mathrm{LD},G,a}$ et $\theta_{\mathrm{LD},G,a}$ à un générateur de code polyédrique.

Une fois le chunk chargé, il faut décider quelles cases garder. On élimine $\vec{e}$ de $\hat{\mathcal{D}}$:

$$\mathbb{V}_{\mathrm{LD},G,a} = \{(\vec{T}, c, o),\ \exists \vec{e} \text{ t.q. } (\vec{T}, c, o, \vec{e}) \in \hat{\mathcal{D}}\}$$

Les points de $\mathbb{V}_{\mathrm{LD},G,a}$ sont de la forme $(\vec{T}, c, o)$ où $\vec{T}$ est la tuile courante, c le numéro du chunk (qu'on vient de charger), et o est n'importe quel numéro d'offset *valide* du chunk c. Pour chaque numero d'offset $0 \leq \omega \leq \Lambda - 1$, on spécialise $\mathbb{V}_{\mathrm{LD},G,a}$ en ajoutant la contrainte $o = \omega$:

$$\mathbb{V}_{\mathrm{LD},G,a}(\omega) = \{(\vec{T}, c),\ (\vec{T}, c, o) \in \mathbb{V}_{\mathrm{LD},G,a} \wedge o = \omega\}$$

Le polyédre $\mathbb{V}_{\mathrm{LD},G,a}(\omega)$ contient alors tous les itérations (tuile,chunk) $(\vec{T}, c)$ dans lesquelles il faut conserver la donnée à l'offset ω et la verser dans le canal de sortie de $\mathrm{LD}_{G,a}$. On construit donc un *filtre* avec les polyédres $\mathbb{V}_{\mathrm{LD},G,a}(0), \ldots, \mathbb{V}_{\mathrm{LD},G,a}(\Lambda - 1)$ qui s'évalue, pour chaque itération $(\vec{T}, c)$, en un masque binaire qui détermine quelles données du chunk conserver.

Exemple (suite). Le domaine d'itération obtenu pour $\mathrm{LD}_{G,a}$ est $\mathcal{D} = \{(I'_1, I'_2, i),\ 1 \leq i \leq N - 2 \wedge 2(I'_2 - I'_1) \leq i < 2(I'_2 - I'_1 + 1)\}$. Avec des chunks de taille $\Lambda = 2$, et en supposant $@a[i] = 0 + 1.i = i$ (adresse de base:0, taille d'une case:1), on a:

$$\hat{\mathcal{D}} = \{(I_1, I_2, c, o, i),\ 2c \leq i < 2(c+1) \wedge o = i - 2c \wedge (I_1, I_2, i) \in \mathcal{D}\}$$

En éliminant les variables o et i de $\hat{\mathcal{D}}$, on obtient le domaine d'itération:

$$\mathcal{D}_{\mathrm{LD},G,a} = \{(I_1, I_2, c),\ 2(I_1 - I_2) \leq c - 1 \leq 2(I_2 - I_1) \wedge c \geq 0 \wedge 2c \leq N - 2\}$$

qui peut être parcouru en appliquant la même technique (First() et Next()) que précédement. Après projection, on obtient:

$$\begin{aligned}
\mathbb{V}_{\mathrm{LD},G,a}(0) = \{(I_1, I_2, c),\ & c \geq \max\{I_2 - I_1, 1\} \wedge 2c \geq 1 \wedge \\
& 2c \leq \min\{2(I_2 - I_1) + 1, N - 2\}\} \\
\mathbb{V}_{\mathrm{LD},G,a}(1) = \{(I_1, I_2, c),\ & 0 \leq c \leq I_2 - I_1 \wedge 2c \geq 1 \wedge \\
& 2c \geq \max\{2(I_2 - I_1) - 1, 0\} \wedge 2c \leq N - 3\}
\end{aligned}$$

□

c) Allocation des canaux d'entrée/sortie Une fois produit les processus LD et ST pour chaque groupe de tuile G, il faut dimensionner et allouer leurs canaux. La taille des canaux doit permettre de réaliser simultanément les entrées/sorties et le calcul. Pour les canaux des LD, on veut pouvoir charger les données pour la tuile $T+1$ pendant que les données pour le calcul de la tuile T sont lues. Pour les canaux des ST on veut pouvoir écrire le résultat du calcul de la tuile $T+1$ pendant que les résultats de la tuile T sont sauvegardés en mémoire centrale.

On construire donc un ordonnancement θ qui formalise cette exécution pipelinée. Soit S une affectation de G, et $\theta(S, \vec{i}) = (\vec{u_1} I_1 \ldots \vec{u_d} I_d \vec{u_{d+1}})$ son ordonnancement. Par définition les affectations de G ont des ordonnancements *DPN-compatibles*, elles partagent donc le même préfixe $(\vec{u_1} I_1 \ldots \vec{u_d} I_d)$. On modifie alors les ordonnancements de la façon suivante:

- Un *processus* $\mathrm{LD}_{G,a}$ reçoit pour ordonnancement $(\vec{u_1} I_1 \ldots \vec{u_d} I_d)$.
- Un *processus de calcul* de G voit son ordonnancement modifié: $\theta(S, \vec{i})[n_d^S] := I_d + 1$.
- Un *processus* $\mathrm{ST}_{G,a}$ reçoit pour ordonnancement $(\vec{u_1} I_1 \ldots \vec{u_d} I_d + 2)$.

Les buffers sont alors alloués comme dans la section précédente, avec ces nouveaux ordonnancements.

d) Synchronisation des canaux des processus LD et ST On procéde comme dans la sous-section 4.1, en construisant un programme producteur/consommateur LD/calcul pour les canaux LD, et calcul/ST pour les canaux ST. Muni de l'ordonnancement construit en c), on calcule les ensembles de dépendance $\mathcal{D}^{\text{flot}}_{P,C}$ et $\mathcal{D}^{\text{anti}}_{C,P}$, et on construit les nouvelles unités de synchronisation en appliquant la technique décrite dans la sous-section 4.1.

e) Synchronisations entre processus LD et ST On pourrait laisser les processus LD et ST se synchroniser par les canaux. Le résultat serait catastrophique en terme de performances, puisque les différentes requêtes mémoire s'entrelaceraient, provoquant des changements de ligne mémoire fréquents et, au final, une mauvaise localité des données. On procède donc comme suit:

- On se donne un ordre entre les processus LD du groupe de tuile (p. ex.: $\text{LD}_{G,a}$ puis $\text{LD}_{G,b}$), puis entre les processus store.
- On commence l'exécution du premier processus LD_a. Quand la derniere dimension tuilée I_d change, on a fini de charger le tableau a pour la tuile. On passe alors au processus LD suivant.
- Si on est déjà sur le dernier LD, on passe au premier ST. Le processus ST est *non-bloquant*: s'il n'a rien à lire (son canal est vide), on repasse au premier LD, et on itère.

Cette dernière étape autorise le double-buffering.

Considérons l'exécution d'une bande avec 4 tuiles, qui ont respectivement besoin des données d_0, d_1, d_2, d_3. Avec ce schéma de synchronisation, on observe l'ordonnancement suivant:

T	LD	C	ST
0	d_0	–	–
1	d_1	d_0	–
2	–	d_1	d_0
3	d_2	d_1	–
4	–	d_2	d_1
5	d_3	d_2	–
6	–	d_3	d_2
7	–	–	d_3

En 0 et en 1, il n'y a encore rien à écrire, juste des loads. Si le ST attendait que son buffer soit rempli, il y aurait une séquentialisation $\text{LD}_0; C_0; \text{ST}_0; \text{LD}_1; C_1; \text{ST}_1; \ldots$. Ici, avec des ST non bloquants et des canaux correctement dimensionnés, on s'autorise une exécution pipelinée. En 1, le processus de calcul lit d_0, et produit un résultat dans le canal du processus ST. En 2, ST lit le résultat, et le sauvegarde vers la mémoire distante. En même temps, le processus de calcul lit la donnée d_1 dans le canal de processus LD. En 3, le processus ST passe la main au processus LD. L'exécution se poursuit jusqu'à ce que toutes les données aient été consommées.

f) Synchronisations entre groupes de tuiles Comme l'ordonnancement est DPN-compatible, les bandes peuvent s'exécuter en séquence. On restreint alors le programme à toutes ses affectations tuilées, et on construit l'automate de contrôle correspondant. Le résultat est une fonction $First() : . \to \Omega(P)$ qui donne la première opération à exécuter, et une fonction de transition $Next : \Omega(P) \to \Omega(P)$ qui retourne l'opération suivante. On prend alors le quotient de ces fonctions par $\sqcup\!\sqcup$ en procédant comme suit. Dans $First()$ et dans $Next(S, \vec{i})$, on remplace chaque clause $\vec{i} \in D_k : (T_k, (u_k(\vec{i})))$ par:

- Rien (clause supprimée), si S et T_k sont dans le même groupe de tuiles ($Cl_{\sqcup\sqcup}(S) = Cl_{\sqcup\sqcup}(T_k)$).
- $\vec{i} \in D_k : Cl_{\sqcup\sqcup}(T_k)$, sinon.

Le résultat est une fonction First() qui donne le premier groupe de tuile à exécuter. Et une fonction $Next(G, \vec{i})$, qui étant donné un groupe de tuiles courant G, dont l'itération courante est $\vec{i}$, indique à quelle condition sur $\vec{i}$ il faut interrompre l'exécution du groupe de tuile courant et passer à un autre groupe de tuile. Il est inutile de passer le vecteur d'itération $u_k(\vec{i})$ au groupe de tuile destination, ses processus étant à jour avec les bonnes valeurs (héritées de sa précédente exécution).

Exemple (suite). Le DPN final obtenu après optimisation des entrées/sorties est donné en annexe 1. □

4.3 Parallélisme

Notre schéma de compilation produit un DPN qui exécute en séquence les instances successives des groupes de tuiles du programme. Le seul parallélisme qui existe est le parallélisme pipeliné entre les processus d'un groupe de tuiles. Cette sous-section présente une transformation simple qui permet d'augmenter à volonté le parallélisme d'un DPN, tout en préservant le schéma pipeliné d'entrée/sortie décrit dans la sous-section précédente.

Dans un groupe de tuile avec au moins deux hyperplans de tuilage $\vec{H}_{d-1}$ et $\vec{H}_d$, il suffit de diviser la bande en sous-bandes en faisant coulisser $\vec{H}_{d-1}$. Chaque sous-bande peut alors s'exécuter en parallèle, modulo des synchronisations de canaux pour garantir les dépendances de flot. En effet:

- Soit $\vec{H}_{d-1}$ ne porte aucune dépendance, et les sous-bandes obtenues sont totalement indépendantes. Aucune synchronisation n'est alors nécessaire.
- Soit $\vec{H}_{d-1}$ porte des dépendances. Dans ce cas, on montre qu'il existe un ordonnancement paralléle pipeliné entre les bandes [14], qui sera spontanément observé par le DPN, dont l'exécution suit directement le flot des données.

Il suffit de spécifier pour chaque groupe de tuile G un nombre π_G de sous-bandes paralléles, et de dupliquer les processus de calcul et les canaux π_G fois. Bien sûr, π_G doit diviser la hauteur ℓ d'une tuile dans la direction de $\vec{H}_{d-1}$. La difficulté principale est de dupliquer les canaux et de construire un multiplexage/démultiplexage correct de sorte que le réseau reste un DPN. Les processus Ld et St, eux, ne sont pas changés. Cette transformation n'affecte donc pas l'optimisation des entrées/sorties décrite dans la section précédente.

Pour chaque processus *de calcul* P du groupe de tuile G, on procéde comme suit:

- *Dupliquer* P en π_G instances: $P_1, \ldots, P_{\pi_G}$. Si $\mathcal{D}_P$ est le domaine d'itération de P, le domaine d'itération de chaque P_k est:

$$\mathcal{D}_{P_k} = \{\vec{i},\ \vec{i} \in \mathcal{D}_P \wedge \ell I_{d-1} + i\ell/\pi_G \leq \vec{H}_{d-1} \cdot \vec{i} < \ell I_{d-1} + (i+1)\ell/\pi_G$$

  Les *canaux écrits* sont dupliqués de sorte que si P écrit dans les canaux $c_1, \ldots, c_n$, P_k écrit dans les canaux $c_1^k, \ldots, c_n^k$. Les canaux dupliqués $c_i^1, \ldots c_i^{\pi_G}$ sont reliés aux mêmes ports de multiplexage que le canal original c_i. Il faut donc changer le multiplexage pour lire la donnée dans le bon c_i^k, c'est l'objet de l'item suivant.

Les *canaux lus* sont inchangés. Si un port de multiplexage de P lit un canal c, le même port de multiplexage du processus dupliqué P_k lit le même canal c. Il faudra dupliquer les canaux d'entrée et le démultiplexage des processus producteurs de P. C'est l'objet du troisième item.

- Pour chaque clause de multiplexage dans G, qui lit un canal c_i produit par un processus de calcul P (dupliqué) $\vec{i} \in D : c_i[u(\vec{i})]$,

  Remplacer la clause par les π_G clauses (pour $k = 1, \pi_G$):

  $$\{\vec{i} \in D \wedge u(\vec{i}) \in \mathcal{D}_{P_k}\} : c_i^k[u(\vec{i}]$$

  De cette façon, le multiplexage est corrigé.

- A ce stade, le réseau est correct, mais ce n'est plus un DPN. En effet, le premier item ne duplique pas les canaux d'entrée pour les processus P_k. Tous les processus P_k lisent donc dans les mêmes canaux. Il reste donc à effectuer cette duplication pour obtenir un DPN correct.

  Pour chaque clause de multiplexage de P_k: $\vec{i} \in D : c[u(\vec{i})]$ qui lit un canal c produit par un processus Q, faire:

  - *Remplacer* le canal c par un nouveau canal c^k (réplique de c).
  - *Ajouter* au démultiplexeur de Q, la clause $\vec{j} \in u(\mathcal{D}_{P_k}) : c^k[\vec{j}]$.
  - *Supprimer* la clause de démultiplexage originale (qui écrit dans c).

Enfin, il faut redimensionner les canaux compte tenu des nouveaux domaines d'itération. On applique pour celà la méthode d'allocation de canaux décrite dans la sous-section 4.1.

Exemple (suite). Supposons que l'on souhaite dupliquer le DPN avec $\pi_G = 2$. Après la première étape, on obtient le DPN transformé donné en figure 3. Le domaine d'itération de S_1 est B_1. Celui de S_2 est B_2. On commence par corriger le multiplexage. Considérons par exemple le multiplexeur de T_1. On ajoutant la condition $u(t,i) = (t,i) \in B_2$ à l'arc qui vient de c^2. Comme $(t,i) \in B_1$ et $B_1 \cap B_2 = \emptyset$, la clause est toujours fausse, et donc l'arc peut être retiré. De façon analogue, on retire l'arc de c^1 vers T_2. Il reste ensuite à dupliquer entrant dans S_1 et S_2 avec les clauses de démultiplexage adéquates. Le DPN final est donné en annexe 3. □

5 Travaux liés

Les réseaux de processus sont des modèles d'exécution paralléle naturels dans la mesure où ils expriment le flot des données, et par là, les contraintes de précédence entre tÃ¢ches. Ainsi, un réseau de processus est une représentation intermédiaire naturelle pour un compilateur paralléliseur, et c'est dans ce contexte qu'on trouve la plupart des travaux sur les réseaux de processus.

Un *réseau de processus de Kahn* (KPN, Kahn Process Network) [9] comporte un ensemble de processus connectés par des FIFO. Chaque FIFO possède un unique producteur et unique lecteur, ce qui assure le déterminisme du réseau, à condition que les fonctions calculées par les processus soient monotones et continues (au sens de la topologie de Scott). Les réseaux de processus de Kahn sont donc réguliers, et sont une forme particulière de DPN dans laquelle les canaux ne peuvent être que des FIFO.

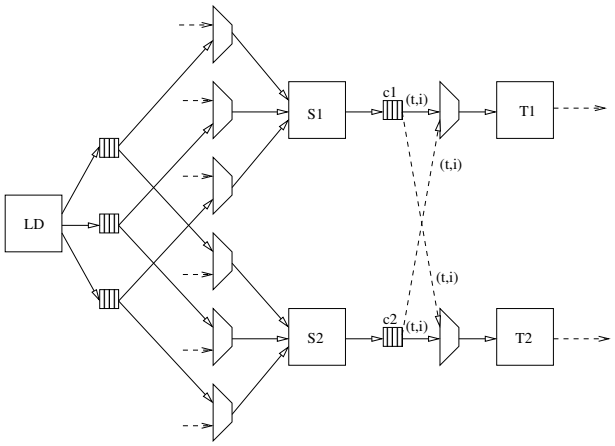


Figure 3: Jacobi 1D: DPN parallèle, étape 1

Un *réseau de processus polyédrique* (PPN, Polyhedral Process Network) [12, 13] est une forme particulière de KPN compilée à partir d'un programme à contrôle statique. La technique de compilation conduit à analyser les motifs de communications entre processus. Lorsqu'un consommateur lit les données dans leur ordre de production, une FIFO est produite directement. Sinon, les données sont stockées dans une mémoire tampon jusqu'à ce qu'elles soient lues. La synchronisation du canal interne se fait avec un masque de bits. Les résultats expérimentaux présentés dans la thèse d'Alexandru Turjan montrent que dans la plupart des application considérées, la plupart des communications se font dans l'ordre, et peuvent donc s'implémenter par une FIFO. A ce jour, aucun mécanisme de tuilage ni d'optimisation des entrées/sorties n'a été décrit pour les PPN.

Un *réseau de processus communiquant régulier* [7] (CRP, Communicating Regular Process) est un réseau de processus régulier dans lequel chaque processus écrit un seul canal en assignation unique (chaque case du canal n'est écrite qu'une fois). Par conséquent, un canal peut être accédé par plusieurs processus. Aussi longtemps que les canaux ont une taille infinie, les CRP sont déterministes et garantis sans interblocage (s'ils traduisent un programme à contrôle statique). Il en va tout autrement dès qu'on dimensionne les canaux. Comme la condition DPN (un seul écrivain, un seul lecteur par canal) n'est pas nécessairement vérifiée, une synchronisation complexe devrait être construite entre le producteur et les consommateurs d'un canal. A ce jour, aucune technique de synchronisation n'a été décrite pour les CRP. De plus, permettre plusieurs lectures simultanées sur un canal est irréaliste dans le domaine de la synthèse de circuits, ce qui rend ce modèle difficilement utilisable. En somme, les CRP peuvent être vus comme un modèle théorique, éloigné des considérations pratiques de la synthèse de circuits. Aucune approche de tuilage et d'optimisation des entrées/sorties n'existe pour les CRP.

La figure 4 résume les relations entre les différents modèles. Les DPN contiennent les KPN (et donc les PPN, qui sont en cas particulier de KPN), puisqu'ils permettent un ordre d'accès arbitraire aux canaux. Quand le programme est tel que chaque référence admet une unique source, le modèle CRP vérifie la condition DPN. En général, les données peuvent provenir de plusieurs producteurs. Dans tous ces cas, les deux modèles divergent.

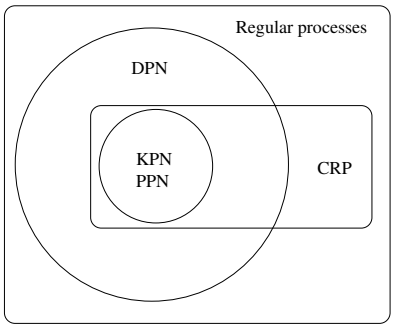


Figure 4: Relations entre plusieurs modèles de réseaux de processus

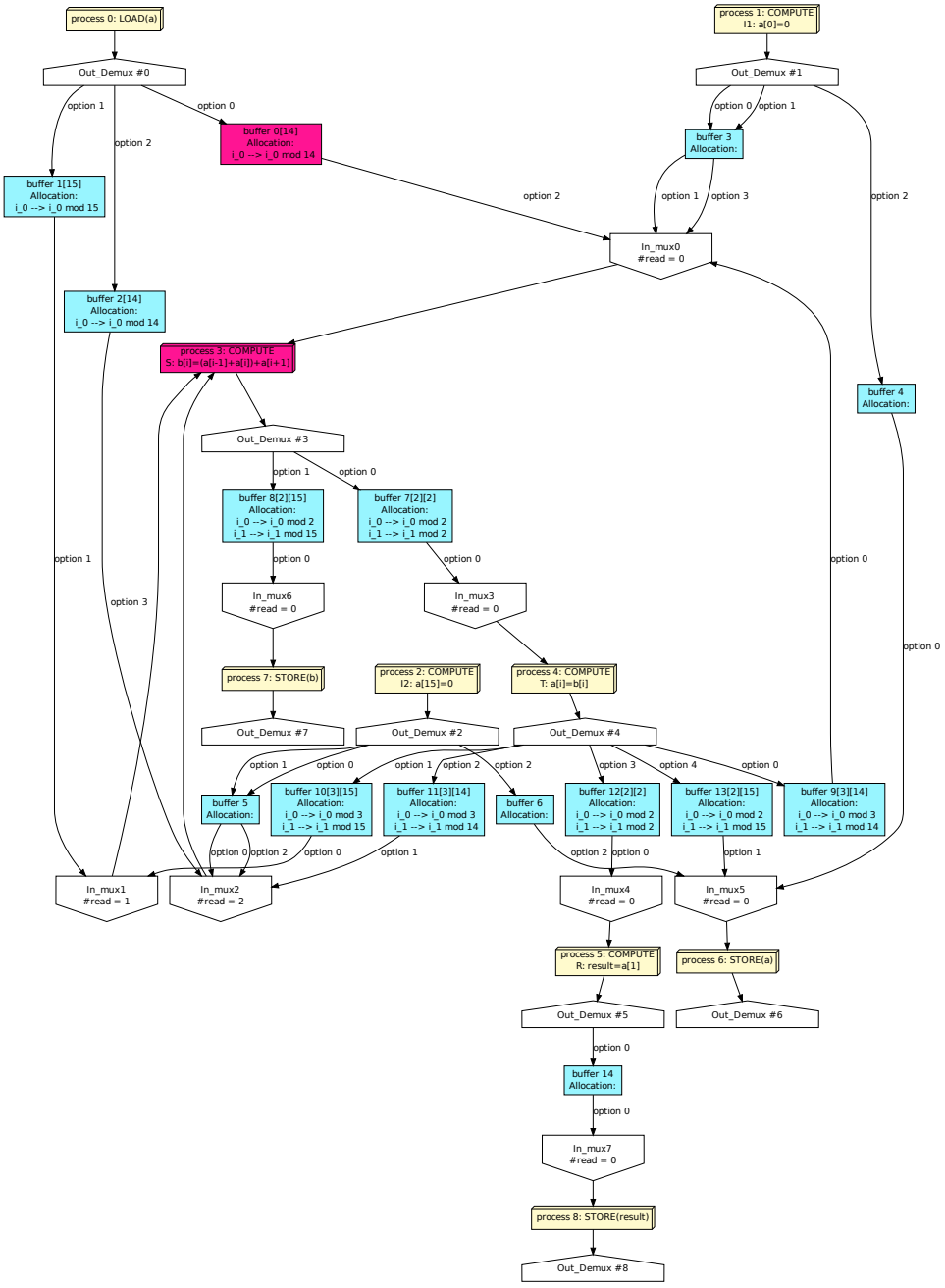
6 Conclusion

Dans ce rapport, nous avons introduit un nouveau modèle de réseaux de processus adapté à la synthèse de circuits, le modèle DPN. Nous avons démontré qu'un schéma de compilation valide produit un DPN déterministe garantit sans interblocage, sans avoir recours à des techniques de synchronisation compliquées entre processus. Seuls les canaux doivent être synchronisés, et nous donnons une technique simple pour y arriver.

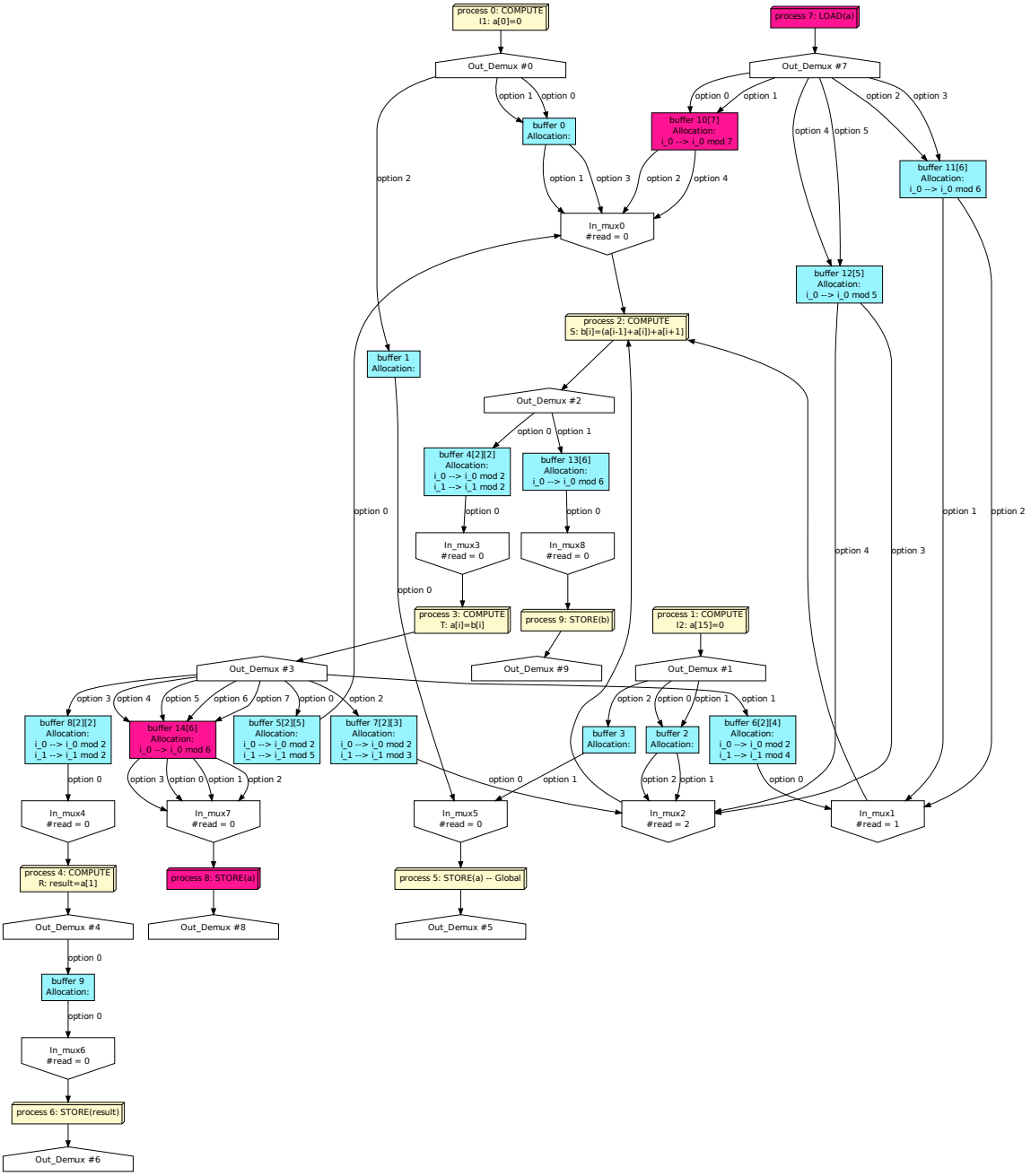
Nous avons également présenté une technique de compilation complète d'un programme à contrôle statique vers un DPN. Plusieurs post-passes d'optimisation sont présentées. D'une part, nous montrons comment optimiser les entrées/sorties. En particulier, nous montrons comment tuiler un DPN, et comment compiler les entrées/sorties intermédiaires de façon à optimiser la réutilisation des données en interne, et à cacher les latences de communication par les calculs. D'autre part, nous montrons comment un DPN peut être parallélisé, sans remettre en cause l'optimisation des entrées/sorties. L'ensemble de ces techniques ont été implémentées dans le compilateur Dcc (DPN C Compiler).

Dans le futur, plusieurs améliorations sont possibles. Tout d'abord, un nombre important de canaux est produit. En tirant profit des dépendences d'entrée entre références, beaucoup de canaux pourraient être remplacés par un seul canal, suivi de plusieurs registres à décalage montés en série, à la sortie desquels se trouveraient les données pour ces références. Cette technique est en cours de conception. Ensuite, il est envisageable d'exécuter plusieurs groupes de tuile en parallèle, à condition de revoir le schéma de synchronisation entre les processus d'entrée/sortie, notamment pour préserver leur localité. Enfin, il est souhaitable de gérer des programmes plus généraux que les programmes à contrôle statique. Comme pour les approches polyédriques classiques, il est possible de cacher la partie irrégulière dans les fonctions calculées par les processus. La difficulté reste de traiter les boucles `while`, notamment de borner assez finement leur nombre d'itérations.

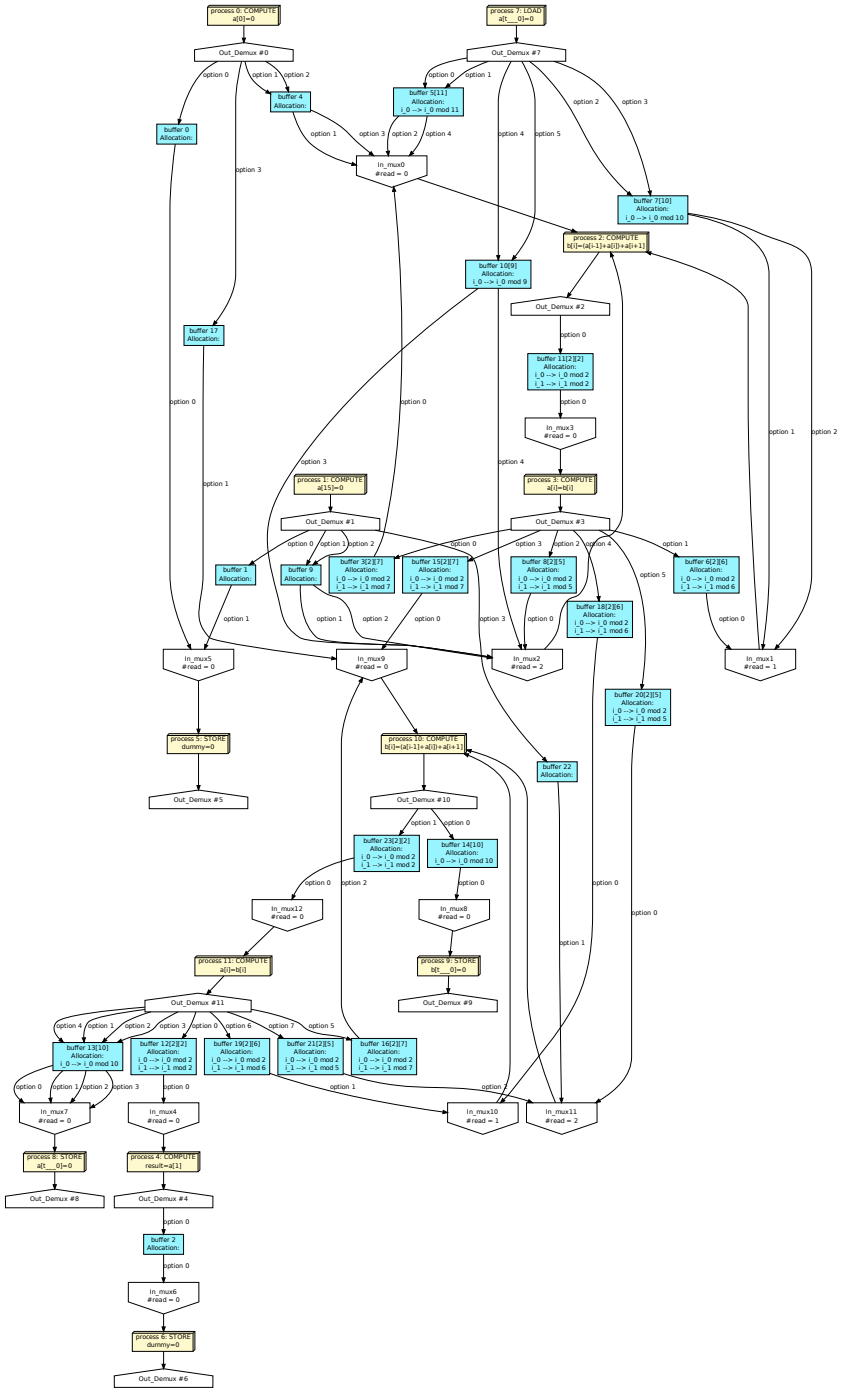
Annexe 1. DPN simple



Annexe 2. DPN après optimisation des entrées/sorties



Annexe 3. DPN après optimisation des entrées/sorties et parallélisation



Références

[1] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for fpga. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[3] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Fpga-specific synthesis of loop-nests with pipeline computational cores. *Microprocessors and Microsystems*, 2012. A paraÃ®tre.

[4] Pierre Boulet and Paul Feautrier. Scanning polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998.

[5] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design and Test*, 2011.

[6] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, 2005.

[7] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, October 2006.

[8] Paul Feautrier and Christian Lengauer. *The Polyhedron Model*. Springer, 2011.

[9] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, 1974.

[10] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

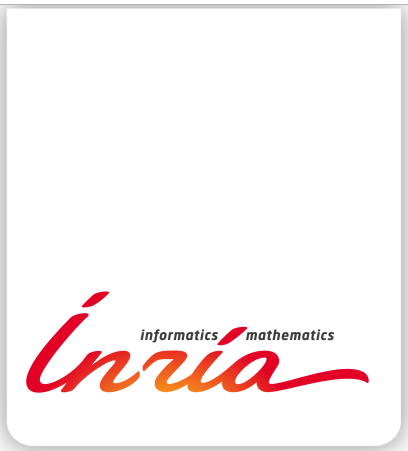
[11] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[12] Alexandru Turjan. *Compiling Nested Loop Programs to Process Networks*. PhD thesis, Universiteit Leiden, 2007.

[13] Sven Verdoolaege. *Polyhedral Process Networks*, pages 931–965. Handbook of Signal Processing Systems. 2010.

[14] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer, 2000.

Table des matières



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399