

numéro d'ordre : 3064

Thèse présentée pour obtenir le titre de Docteur de l'Université des Sciences et Technologies de Lille, spécialité Informatique

Principe de compilation d'un langage de traitement de signal

Julien SOULA

Laboratoire d'Informatique Fondamentale de Lille
Université des sciences et Technologies de Lille

Soutenue le 20 décembre 2001

Après avis de Rumen ANDONOV et Isaac SCHERSON

Devant la commission d'examen formée de

Rumen ANDONOV
Vincent CORDONNIER
Alain DEMEURE
Jean-Luc DEKEYSER
Philippe KAJFASZ
Philippe MARQUET
Isaac SCHERSON

Remerciements

Table des matières

Introduction	9
1 Le traitement de signal systématique	11
1.1 Les systèmes réactifs synchrones	11
1.1.1 ESTEREL	12
1.1.2 LUSTRE	14
1.1.3 PTOLEMY	15
1.2 Le traitement de données	17
1.2.1 Le parallélisme de données	17
1.2.2 Les langages	20
1.3 Conclusion et application au TS	23
2 L’environnement de programmation ARRAY-OL et ses alternatives	25
2.1 Un atelier de développement d’applications TS	26
2.2 Le langage ARRAY-OL	26
2.2.1 Le modèle global	26
2.2.2 Le modèle local	28
2.2.3 Action élémentaire d’une tâche	31
2.2.4 La hiérarchie	31
2.2.5 Contraintes du langage	33
2.2.6 Le formalisme matriciel	34
2.2.7 Représentation textuelle	35
2.3 Compilation d’un code ARRAY-OL	38
2.3.1 Schéma d’implantation séquentiel direct	38
2.3.2 Points sensibles de la génération de code	40
2.3.3 Court-circuit vs. pavage incrémental	42
2.3.4 Limitations du schéma de compilation	46
2.4 Les environnements existants	48
2.4.1 PEI	48
2.4.2 ALPHA	49
2.4.3 Méthodologie pour le <i>HLS</i>	51
2.4.4 PIPS	52
2.4.5 PLC pour la compilation d’application TS	53

2.4.6	SUIF	54
2.4.7	Conclusion	55
2.5	Modèles linéaires et outils	55
2.5.1	Caractéristiques du domaine d'étude	55
2.5.2	Les modèles formels	56
2.5.3	Les outils existants	58
2.5.4	Conclusion	59
3	Modification du schéma d'exécution par hiérarchisation	61
3.1	Nouvelle approche d'exécution	61
3.1.1	Schéma d'exécution direct	61
3.1.2	Schéma d'exécution dynamique et parallélisme de l'exécution . . .	62
3.1.3	Hiérarchisation	62
3.2	Un formalisme approprié : les ODT	64
3.2.1	Notations	64
3.2.2	Les opérateurs élémentaires	65
3.2.3	L'ensemble des ODT	67
3.2.4	ODT miroirs	68
3.2.5	Application d'un ODT	69
3.3	ODT et ARRAY-OL	70
3.3.1	Représentation d'une tâche ARRAY-OL par les ODT	70
3.3.2	Forme ODT d'une hiérarchie	72
3.3.3	Quelques propriétés sur les ODT	74
3.3.4	ODT exact	76
3.3.5	Inversion d'un ODT exact	77
3.3.6	Modulo redondant	77
3.4	Transformation fondamentale : la fusion	78
3.4.1	Orientation de transformation de l'expression des dépendances . .	80
3.4.2	Dépendances fractionnaires	81
3.4.3	Pavage entier	84
3.4.4	Regroupement des motifs	84
3.4.5	Ajustage et décalage entier	86
3.4.6	Dépendances finales	88
3.4.7	Mise en forme hiérarchique	90
3.4.8	Cas d'une séquence de tâches quelconque	91
3.5	Vers une plate-forme de transformation	94
3.5.1	Évaluation de la fusion	94
3.5.2	Recouvrement de motifs et de calculs	97
3.5.3	Extension de la fusion élémentaire	99
3.6	Stratégie de compilation	102
3.6.1	Dénombrement	102
3.6.2	Hiérarchisation de la Veille Large Bande	103
3.6.3	Critères objectifs	108

3.6.4	Expérience de compilation parallèle	112
3.7	Conclusion	116
4	Transformations visuelles dans l'environnement GASPARD	117
4.1	Introduction	117
4.2	L'environnement GASPARD	118
4.3	L'architecture logicielle	120
4.3.1	Les langages et outils utilisés	120
4.3.2	Arbre syntaxique abstrait	121
4.3.3	Le serveur	122
4.4	Exemple de transformation	123
4.5	Conclusion	127
	Conclusion	129
A	Sources ARRAY-OL des différentes formes de la Veille à Large Bande	133
A.1	VBL infinie	133
A.2	VBL infinie hiérarchisée	136
A.3	VBL utilisée pour les tests de performance	140

Introduction

La construction de Sonars Nouvelle Génération mise en œuvre par Thomson Marconi Sonar (TMS) repose pour ce qui est de la partie calculatoire sur la spécification et l'exécution d'applications de traitement de signal. Une chaîne complète de traitement sonar se décompose schématiquement en une partie d'acquisition de signaux par des capteurs, suivie de la transformation de ces signaux afin d'en extraire des données (traitement de signal, TS), elle-même prolongée par une partie d'exploitation des données qui permet le suivi des trajectoires des objets détectés (traitement de données).

Afin de répondre aux besoins de spécification, de standardisation et d'efficacité de la partie traitement de signal, TMS a développé un langage orienté traitement de signal : ARRAY-OL (Array Oriented Language) [DLB⁺95]. La compilation de ce langage vise autant les stations de travail Unix, principalement dans le but de mettre au point les applications, que des machines multiprocesseurs dédiées à ARRAY-OL et destinées à être embarquées. Seul un sous-ensemble de ce langage était, au départ de ce travail, supporté par la version Unix du compilateur.

L'objet des travaux présentés dans cette thèse concerne l'évaluation de ce compilateur, la proposition d'optimisations du code généré et l'extension vers une version « full » ARRAY-OL sur diverses architectures.

Pour obtenir une chaîne de compilation adaptée à plusieurs classes d'architecture, nous proposons de mettre en place un système de transformation d'un programme ARRAY-OL en un programme ARRAY-OL. Ces transformations sont indispensables pour certaines applications, par exemple celles qui manipulent des tableaux ayant une dimension infinie. Elles sont nécessaires pour orienter le compilateur dans ses tâches de placement et d'ordonnancement sur des machines parallèles à mémoires partagées ou distribuées. En effet, ARRAY-OL ne propose pas, à l'inverse d'autres langages data-parallèles, des extensions ou directives de répartition de données. Un compilateur 100% ARRAY-OL est réalisé pour telle ou telle machine, c'est en amont de celui-ci par des transformations de codes sources que le regroupement des données, et donc leurs distributions, seront explicitées. À partir de ces nouvelles spécifications le compilateur se doit de proposer un ordonnancement en fonction de la machine ciblée.

Les transformations mises en œuvre dans notre environnement reposent sur une description formelle des dépendances de données basée sur un formalisme adéquat : les

ODT (opérateurs de distribution de tableaux). À partir de quelques transformations de base, associées à des métriques d'efficacité, nous proposons des stratégies de transformation. Ces transformations ont été intégrées à un environnement de programmation visuelle : GASPARD. Une démonstration de ces transformations sur un cas typique permet de valider la démarche.

Le chapitre 1 présente divers systèmes dédiés au traitement de signal. Ils sont basés sur des systèmes réactifs synchrones soit impératifs soit déclaratifs. À l'évidence le domaine du traitement de signal intensif tel qu'il est défini dans le modèle ARRAY-OL ne peut s'exprimer directement dans ces langages qui ne proposent pas de constructeur data-parallèle. D'un autre côté les langages data-parallèles traditionnels ne supportent pas directement les contraintes liées au temps réel.

C'est pourquoi dans le chapitre 2, nous introduisons le modèle ARRAY-OL et étudions les difficultés liées à sa compilation sur différentes classes d'architecture. Après une présentation du compilateur et des optimisations que nous y avons intégré, plusieurs systèmes de transformation de codes seront présentés. Aucun d'entre-eux ne permet effectivement de conserver un code résultat respectueux du modèle initial. Il n'est alors pas possible de produire du code ARRAY-OL ce qui permettrait de réduire la portabilité entre différents types d'architecture à juste une paramétrisation différente des transformations appliquées.

Le chapitre 3 correspond à notre contribution principale dans cette thèse. Il concerne les transformations proprement dites. Nous présentons les ODT, support formel d'expression des dépendances de données dans une application ARRAY-OL. À partir de cette formalisation nous proposons les transformations de base qui permettront ensuite de proposer des stratégies de transformation par insertion de niveaux hiérarchiques en ARRAY-OL.

Le chapitre 4 reprend ces transformations mais sous l'aspect visuel. À partir d'une application, nous démontrons comment créer une application ARRAY-OL sans aucune contrainte de syntaxe. Puis pour permettre la compilation d'une application qui manipule des tableaux de taille infinie (sur une machine à mémoire finie) nous transformons le code source depuis l'environnement et toujours dans un contexte visuel.

Enfin pour conclure, des perspectives liées aux collaborations en cours entre le LIFL, TMS, Thales et Esterel Technologies seront explicitées.

Chapitre 1

Le traitement de signal systématique

Le *traitement de signal* convertit les données captées en des données exploitables par la phase de traitement de données. La plupart du temps, le traitement de signal correspond à une phase de traitement *data-parallèle* sur les signaux captés : ceux-ci sont soumis à la même loi de bruitage.

On assimile le *traitement de données intensif* à une réduction de données. En effet, il s'agit le plus souvent d'extraire les informations pertinentes d'un ensemble important de données. Il comprend les opérations de collecte, d'entrée, de transmission, d'édition, d'enregistrement, de tri, de sortie... des données.

On identifie deux phases au traitement de signal :

- les interactions extérieures : les langages synchrones réactifs temps-réel permettent de définir les contraintes temporelles ;
- l'expression de la partie transformationnelle : elle correspond à la description de l'exécution calculatoire. Elle peut comprendre un modèle de spécification d'architecture, un système de description d'algorithmes.

Nous verrons que les deux phases sont souvent explicitées par le biais d'outils orthogonaux. Le programmeur est amené à gérer les deux types de transcription en même temps. Le couplage de ces deux familles n'est pas usuel. Le langage ARRAY-OL proposé par TMS a été construit pour pallier cette difficulté.

Dans ce chapitre nous présentons les exemples les plus représentatifs des langages synchrones. Puis les solutions pour le traitement des données proprement dit seront explicitées dans le cadre du paradigme de programmation à parallélisme de données.

1.1 Les systèmes réactifs synchrones

Les systèmes réactifs synchrones, par opposition aux systèmes transformationnels, se caractérisent par une interaction permanente avec un environnement extérieur. Ils doivent donc réagir immédiatement aux événements qui se produisent. De tels sys-

tèmes se différencient des systèmes interactifs classiques en ce que, dans ces derniers, le temps de réaction est imposé par le système et non par l'intervention.

Les systèmes temps-réel sont des systèmes réactifs avec en plus de fortes contraintes de temps sur la production de réponses. Dans le cas des applications de traitement de signal, ils doivent réagir immédiatement aux flots de données entrants et produire ses résultats dans le temps imparti pour alimenter la suite de la chaîne de traitement.

Il existe plusieurs moyens spécifiques pour décrire de tels systèmes :

- à base de graphes : Grafcet [AP93], Petri [UJ98], StateChart [Har87], PTOLEMY [JBM94]
- par des langages impératifs : ESTEREL [BG92]
- par des langages déclaratifs : LUSTRE [HCRP91] (horloge universel), SIGNAL [PTMC91] (multi horloge)

Il n'est pas rare de trouver des systèmes réactifs écrits directement en assembleur et ceci pour des raisons d'efficacité. D'un autre côté, plusieurs langages classiques (C, ADA) ont été associés à des systèmes d'exploitation (UNIXRT) et des systèmes de communications temps-réel mais cela au détriment de l'efficacité et avec la nécessité, pour le programmeur, d'intervenir au niveau système.

L'intérêt des langages spécialisés est, en premier lieu, de simplifier l'expression des applications mais également de proposer des cadres formels permettant de prouver le déterminisme et la prédictabilité d'une implémentation à partir de sa spécification.

1.1.1 ESTEREL

► Description

Le langage ESTEREL [BG92] fait figure de pionnier dans son domaine, il a été défini par G. Berry, S. Miosan et J.-P. Rigault en 1983. ESTEREL est un langage synchrone impératif, orienté vers le contrôle de flot : les structures qu'il propose permettent de décrire les interactions d'événements (attente de signal, interruption de traitement...). Cette approche est opposée à celle des langages comme LUSTRE et SIGNAL qui mettent en avant les relations entre flots de données (sous forme déclarative) puis en déduisent le contrôle.

Les variables du langage ESTEREL sont, d'une part celles qui interagissent avec l'extérieur c'est-à-dire les signaux d'entrée et de sortie, et d'autre part les variables internes classiques. Leurs espaces de valeurs peuvent être booléens ou numériques.

Les principales structures du langage sont l'attente de signal (`await`), l'émission de signal (`emit`), la séquence (`;`), la concurrence (`||`), l'interruption (`abort . . . when`) et la boucle (`loop . . . each`).

Le module ABRO illustre quelques-unes de ces structures (figure 1.1). Ce module émet le signal O dès qu'un signal A et un signal B ont été reçus et répète ce processus à chaque

```
module ABRO
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end
```

FIG. 1.1 – *Module ABRO en ESTEREL*

fois qu'un signal R est reçu.

► Caractéristiques

Il est important de noter que le langage ESTEREL propose de travailler dans un environnement synchrone idéal. Cela signifie qu'il n'y a aucun délai dans l'enchaînement des processus. Ainsi, dans notre exemple, le signal O n'est pas émis après l'arrivée du deuxième signal attendu mais bien en même temps. Ceci rend le code `present O else emit O` contradictoire puisque la présence (i.e. l'émission) de O interdit son émission.

Le langage est évidemment très proches des spécifications par automates ou par machines de MEALLY, la différence essentielle tient à la forme condensée du code. En fait, un automate nécessiterait la spécification de toutes les transitions possibles (A sans B, B sans A, A et B sans R...), ainsi l'ajout, par exemple, d'une entrée C doublerait l'automate alors que le code ESTEREL reste linéaire (on remplacerait seulement la ligne d'attente des signaux par `[await A || await B || await C];`).

Cependant on retrouve bien des automates au final (au format OC [CSP90]) puisque c'est la forme du résultat produit par le compilateur. À partir de ceux-ci, on peut générer du code en C, en Le-Lisp et en ADA. On peut également, et c'est un des principaux objectifs de ce type de langage, utiliser ces automates pour vérifier des propriétés de programmes. En particulier, puisque le langage ne propose pas de boucle de traitement et respecte la condition synchrone, il est possible d'obtenir une véritable spécification du temps-réel.

1.1.2 LUSTRE

► Description

Bien que défini pour le même type d'applications, le langage LUSTRE [HCRP91], créé par P. Caspi, D. Pilaud, N. Halbwachs et J. Plaice en 1987, se distingue du langage ESTEREL par le type de ces variables (des flots de valeurs rythmés par des horloges) et un style de programmation déclaratif.

Un flot est l'association d'une suite de valeurs et d'une horloge dont les cycles indexent les valeurs du flot. Une variable représente donc l'ensemble des valeurs d'un signal au cours du temps. Autant dans ESTEREL la notion de cycle était sous-entendue par l'avènement de signaux (par exemple l'hypothèse d'existence d'un signal *Seconde* pouvait permettre de séquencer un programme), autant dans LUSTRE les horloges représentent implicitement le facteur de synchronisation. Il existe une horloge de référence pour tout le système mais chaque flot peut avoir une sous-horloge de celle-ci. L'horloge d'un signal est elle-même un signal booléen (figure 1.2).

<i>signal X sur l'horloge de référence</i>	↔	(1	2	4	7	9	12	...)
<i>signal booléen B sur l'horloge de référence</i>	↔	(false	true	true	false	false	true	...)
<i>signal X sur l'horloge B</i>	↔	(1	2		4	...)	

FIG. 1.2 – Exemple de signaux constitués de la même suite de valeurs mais sur des horloges différentes

Un programme LUSTRE se caractérise par un système d'équations sur des flots. Hormis les opérateurs fonctionnels classiques (*if . . . then . . . else*, *+*, ***...), les principaux opérateurs structurels de flots sont : (*pre*) qui décale les valeurs d'un cycle ; (*→*) qui permet d'initialiser la première valeur d'un flot avec celle de l'autre ; (*when*) qui permet d'échantillonner un flot sur une horloge définie par un flot booléen ; et enfin (*current*) qui sur-échantillonne le flot sur l'horloge de son horloge en extrapolant par escalier les valeurs non définies. On notera également que pour toutes les opérations non unaires les signaux impliquées doivent avoir la même horloge.

Le module *ABRO* défini pour ESTEREL, peut se coder en LUSTRE (figure 1.3)

Le sous-module *bascule* émule le fonctionnement d'une bascule c'est-à-dire que le signal passe à vrai sur *set*, à faux sur *reset* et conserve son état sinon. Les variables *a'*, *b'* et *o'* indiquent l'arrivée des signaux *a* et *b* et *o* depuis la dernière apparition du signal *r*.

```
node bascule(set, reset: bool) return (b: bool);
let
  b = current( set and not(reset) when (set or reset) );
tel.

node ABRO(a, b, r: bool) return (o: bool);
var a', b', o': bool;
let
  a' = bascule(a,r);
  b' = bascule(b,r);
  o' = not(bascule(o,r));
  o = a' and b' and (true->pre(o'));
tel.
```

FIG. 1.3 – *Module ABRO en LUSTRE*

► Caractéristiques

La phase de compilation d'un programme passe tout d'abord par une vérification de la validité du système. En effet, du fait même de la nature déclarative du langage, le compilateur doit vérifier l'absence de cycle ou de récursion infinie puis retrouver un ordre d'exécution impératif. Le résultat de la compilation est de la même forme que ESTEREL, à savoir des automates au format OC. Cela permet à LUSTRE de partager les mêmes outils de génération de code ou d'analyses de propriétés.

Partant des mêmes applications et aboutissant aux mêmes formats de sortie, la différence entre les deux langages tient donc essentiellement à l'opposition impératif/déclaratif, chacun des styles s'adaptant plus ou moins bien à un algorithme donné.

► SIGNAL

Le langage SIGNAL [PTMC91] est très proche de LUSTRE, les deux langages partagent les mêmes types de variables et le même style déclaratif. Nous ne nous étendrons donc pas sur ce langage. Cependant, SIGNAL diffère de LUSTRE en ce qu'il permet de manipuler les valeurs non définies entre les cycles d'horloge. Cette caractéristique augmente le pouvoir d'expression du langage au détriment d'une complexité accrue des programmes et de la perte de quelques propriétés utilisées par le compilateur.

1.1.3 PTOLEMY

Le projet PTOLEMY [JBM94] a pour but de modéliser et de définir des systèmes concurrents et hétérogènes et de les faire interagir. Il a débuté en 1990 à l'université

de Berkeley par une version écrite en C++. Depuis 1998, il existe une seconde version en JAVA : PTOLEMYII [Dav00].

La description d'une application sous PTOLEMY se fait sous la forme d'un graphe d'acteurs (nœuds d'exécution) reliés par des arcs. L'interprétation des acteurs et des arcs diffèrent suivant le domaine choisi. PTOLEMY en propose un certain nombre :

- CSP (Communicating Sequential Processes)
- CT (Continuous Time)
- DE (Discrete-Events)
- DDE (Distributed Discrete Events)
- DT (Discrete Time)
- FSM (Finite-State Machines)
- PN (Process Network)
- SDF (Synchronous Dataflow)
- SR (Synchronous/Reactive)

Les domaines qui nous intéressent particulièrement sont évidemment DT et SDF. Pour ce dernier, les acteurs sont des tâches qui sont déclenchées dès que leurs arcs entrants sont alimentés par les tâches précédentes. Il existe donc une unité de quantification sur les données véhiculées et l'un des rôles du modèle est de calculer les fréquences de déclenchement des tâches pour permettre d'avoir un flot continu de données (cf. figure 1.4). Il s'agit, en fait, d'une restriction du domaine PN [LP95] qui lui décrit un modèle de passage de messages asynchrones. Ces restrictions permettent de détecter la présence d'interblocages ou de cycle et de définir statiquement des ordonnancements séquentiels ou parallèles. Le domaine DT devrait proposer une extension du domaine SDF en intégrant la notion d'intervalle de temps dans les flots de données mais nous ne nous étendrons pas sur cette partie puisqu'elle n'a pas été encore implémentée.

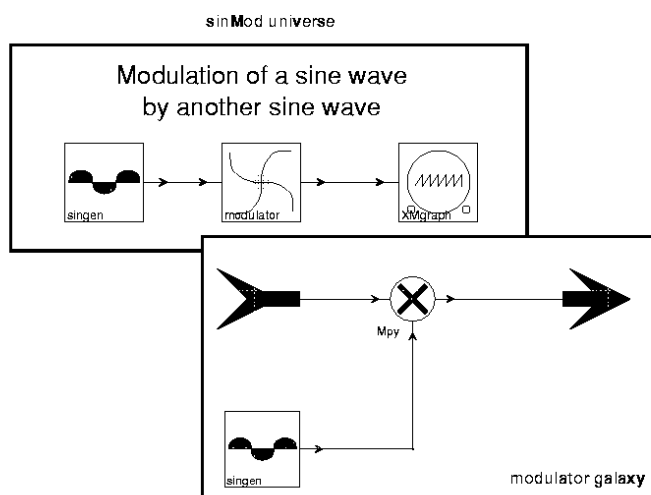


FIG. 1.4 – Exemple d'univers (application) PTOLEMY dans l'étoile (domaine) SDF. Son rôle est de moduler deux signaux et d'afficher le résultat. La cadre supérieur représente le niveau global. Le cadre inférieur décrit la galaxie (sous-application) *modulator*

PTOLEMY permet enfin de raffiner les domaines c'est-à-dire de spécifier plus précisément les définitions des acteurs et des arcs en ajoutant, par exemple, des fonctions de *parsing*, de *printing* et d'exécution propres à l'application développée.

Cette possibilité a été choisie par TMS pour intégrer leur langage ARRAY-OL au domaine SDF qui était particulièrement bien adapté aux spécifications de leur modèle global.

1.2 Le traitement de données

La partie transformationnelle constitue le cœur opératif d'une application. Les objets à traiter et à produire sont généralement des flots de données (provenant par exemple de capteurs) sur lesquels sont appliqués des traitements (FFT, intégration...).

Dans le domaine particulier du traitement de signal, on peut extraire un certain nombre de généralités sur les données et les calculs :

- les structures sont régulières tant au niveau des objets globaux (souvent des tableaux multi-dimensionnels éventuellement infinis) qu'au niveau des structures de calculs (partitionnement régulier des tableaux) ;
- par essence même, les calculs consistent souvent à appliquer des traitements locaux relativement indépendants sur toutes les parties de tableaux. Ils sont donc très bien adaptés au parallélisme de données ;
- les structures sont simples et les calculs répétitifs, la complexité n'est pas dans l'expression de l'algorithme. L'objectif est clairement le traitement numérique ;
- enfin ces applications sont pour la plupart censées s'exécuter sous des contraintes dynamiques (temps réel) aussi bien que physiques (hiérarchie mémoire de machines embarquées).

Nous allons aborder les différents aspects des langages parallèles sous l'éclairage de ces propriétés.

1.2.1 Le parallélisme de données

► Modèle de programmation

Si l'essence même du parallélisme reste toujours d'effectuer des actions en même temps sur des unités d'exécution différentes, on peut distinguer plusieurs classes de parallélisme :

- le **parallélisme de tâche** consiste à faire exécuter par les unités de calcul des actions différentes ;
- un **pipeline** est constitué d'un enchaînement de tâches au travers duquel passe un unique flux de données ;

- pour le **parallélisme de données**, les unités de calculs effectuent toutes la même action mais sur un ensemble de données différentes.

Dans le cas des applications qui nous intéresse, on peut retrouver ces différentes classes de parallélisme. Le parallélisme de tâches et le pipeline sont principalement présents au niveau des enchaînements de tâches (généralement un graphe de tâches *dataflow*) alors que le parallélisme de données se retrouve massivement à l'intérieur des tâches de calculs (calculs d'une FFT ou d'une intégration sur un tableau).

► Modèle d'exécution

La classification de FLYNN propose de différencier le fonctionnement des unités d'exécution sur deux axes orthogonaux selon qu'elles effectuent des instructions différentes ou non (*single/multiple instruction*) et selon qu'elles travaillent sur les mêmes données ou non (*single/multiple data*).

	single instruction	multiple instruction
single data	SISD	MISD
multiple data	SIMD	MIMD

FIG. 1.5 – Tableau de la classification de FLYNN

Évidemment, certains modèles d'architecture se prêtent mieux à certains modèles de programmation. Ainsi le SISD correspond, en fait, au modèle séquentiel alors que le SIMD est plus orienté vers la parallélisme de données et le MIMD mieux adapté aux parallélisme de tâches. (Le MISD est un modèle assez marginal dont l'intérêt n'est pas évident.)

On identifie également un cinquième modèle, SPMD pour *single program multiple data*, qui représente une extension du SIMD en ce que les unités effectuent les mêmes traitements mais à des rythmes différents.

► Modèle mémoire

La classification proposée par FLYNN peut être raffinée pour préciser l'organisation mémoire de machins multiprocesseurs. Les deux principaux modèles sont les mémoires partagées où toutes les données sont dans un espace d'adressage unique accessible par toutes les unités d'exécution, et les mémoires distribuées où chaque unité possède son propre espace d'adressage indépendant les uns des autres. Dans ce dernier cas les échanges se font par des périphériques externes (le réseau, par exemple). Ces deux modèles peuvent également se combiner (des groupes de processeurs partagent la mémoire, l'ensemble étant distribué), on obtient alors une hiérarchie mémoire plus complexe (*clustering*).

Si le modèle à mémoire partagée simplifie la gestion des données et les interactions de tâches, il nécessite aussi l'existence d'un protocole d'arbitrage sur les accès mémoires. Le modèle à mémoire distribuée est de construction plus simple (il «*suffit*» de relier plusieurs machines ensemble) et permet une plus grande intégrité des données mais est très souvent lourdement pénalisé par les échanges de données (coût des communications).

► Évaluation

Dans un modèle d'exécution séquentiel les critères principaux d'évaluation sont le temps d'exécution total du programme, et, dans une moindre mesure, la quantité de mémoire nécessaire. Le moyen d'optimisation principal consiste souvent à améliorer l'algorithme.

En contexte parallèle, ces deux critères restent la finalité de l'évaluation mais on doit rajouter une dimension de mesure qui est celle de la capacité de parallélisme mise en œuvre i.e. le nombre d'unités de calculs. Cela détermine de nouveaux indices d'évaluation : accélération, efficacité et extensibilité.

Le critère le plus naturel consiste à calculer le gain de temps de l'exécution parallèle : l'accélération est égale au rapport du temps séquentiel sur le temps parallèle, elle est évidemment fonction du nombre d'unités de calcul mis en jeu. L'efficacité normalise la fonction d'accélération en la divisant par ce nombre d'unités. Dans les cas très favorables où cette dernière est égale à 1 (« on va N fois plus vite avec N machines ») on dit qu'on a une accélération linéaire. Le plus souvent, elle est sous-linéaire à cause des portions non parallélisables du code (synchronisation) et, dans de très rares cas, elle peut être sur-linéaire (par exemple grâce à une meilleure utilisation des caches).

La mesure d'extensibilité permet de savoir jusqu'à quel point une application donnée bénéficie de l'augmentation des ressources parallèles. Par exemple, un problème constitué de N tâches indépendantes dont chacune est strictement séquentielle aura une très bonne accélération jusqu'à N puis ne sera plus extensible au-delà.

En mémoire partagée, ces mesures sont essentiellement pénalisées par les parties séquentielles de l'algorithme, ainsi que par un mauvais équilibrage des quantités de travail allouées à chaque unité.

En mémoire distribuée, il faut aussi considérer le problème des interactions entre les différentes unités. Ces communications se faisant par des réseaux bien plus lent que les accès locaux à la mémoire, elles se révèlent très souvent les facteurs principaux d'inefficacité. C'est pourquoi il existe de nombreuses études de distribution visant à augmenter la localité des données au sein de chaque mémoire et il n'est pas rare de préférer dupliquer des données et du code pour éviter des communications.

Dans le cas des modèles transformationnels, les tâches constituant le graphe d'exécution sont généralement indépendantes, elles impliquent donc un parallélisme de tâches

parfait (sans communication). Pour les calculs internes aux tâches, il s'agit bien souvent d'appliquer la même fonction à différentes parties d'un tableau (comme effectuer une FFT sur les tranches de temps successives de la sortie d'un capteur). Ce genre de traitement est très fortement *data-parallèle*. Dans les deux cas, la parallélisation ne demande que peu d'efforts.

De plus, toujours dans le contexte donné, la distribution d'une tâche en mémoire distribué ne pose pas trop de difficulté du fait de l'indépendance des motifs de calcul. Par contre la distribution d'une séquence entière de tâches nous confronte de nouveau, à cause de leurs interdépendances, au problème du choix de la distribution pour minimiser les communications.

1.2.2 Les langages

► Introduction

Un langage permet d'exprimer des procédures d'exécution au moyen de concepts de plus ou moins haut niveau. Il existe une pléiade de langages séquentiels ayant tous leur propre syntaxe et certaines caractéristiques distinctives (déclaratif, fonctionnelle, interprété...). Ils ont selon les cas deux motivations principales :

- **simplifier l'écriture du code.** Le langage natif des ordinateurs, s'il est par essence le plus efficace et le moins contraignant, n'est évidemment pas adapté à l'écriture et à la maintenance d'un programme conséquent. C'est pourquoi beaucoup de langage propose des structures de données et de contrôles évoluées qui ne suivent pas forcément le modèle de la machine (concept objet, description déclarative...)
- **permettre un exécution efficace.** Tous les programmeurs n'étant pas des experts, l'expression d'un algorithme dans un modèle plus abstrait permet au compilateur ou à l'interpréteur du langage de trouver une implémentation plus efficace que ne l'aurait fait l'utilisateur.

Les remarques précédentes sont d'autant plus vraies quand on bascule dans le domaine du parallélisme. En effet, il nous paraît indispensable d'avoir un langage adaptée au contexte du traitement de signal : les concepteurs d'applications ne sont pas forcément (et n'ont pas à être) des informaticiens, il leur faut donc un langage qui ne propose pas une syntaxe inutilement compliquée ; de plus, notre travail consiste en la compilation de telles applications, il est alors souhaitable (sinon indispensable) que la structure des algorithmes ne soit pas noyée dans un modèle de spécification trop concret afin que nous puissions la manipuler.

► Généralité sur les langages parallèles

Il existe déjà une série de langages dédiés au parallélisme, quelques uns pour le parallélisme de tâche (ADA [Ame83], OCCAM [JG88]...) mais la plupart pour le parallé-

lisme de données [Mar93].

En ce qui concerne le parallélisme de tâches, les langages et bibliothèques proposés suivent à peu près tous le même schéma en utilisant généralement les possibilités du système sur lequel ils s'exécutent : ils permettent de créer d'autres processus concurrents et ces processus coopèrent par échanges de messages (PVM [Sun90], MPI [MPI94]...) ou bien par passage d'arguments (RPC [Sun88]).

Les véritables différences reposent sur l'intégration du parallélisme de données. Les langages se basent généralement sur la structure de tableaux multi-dimensionnels mais divergent sur les possibilités de manipulations qu'ils permettent et la sémantique d'expression du parallélisme. Ils se démarquent aussi par leur tentative de faciliter l'expression des algorithmes parallèles, la distribution des données (HPF), par leur adéquation au calcul numérique intensif (FORTRAN) ou plus simplement par le fait qu'ils soient interactifs (MATLAB). Bien que quelques uns soient originaux comme OCCAM, la majorité sont en fait des extensions de langages séquentiels proposant des structures ou simplement des directives de données et de contrôles parallèles (C*, HPF).

► Calcul numérique

L'utilisation du parallélisme a été motivée par le besoin croissant en ressources de calculs. Il est donc normal que des langages résolument tournés vers le calcul numérique intensif soient apparus très tôt. On peut citer le plus connu, FORTRAN [ABM⁺92], et son avatar pour le placement distribué, HPF [HPF97]. Les structures qu'ils manipulent se limitent aux tableaux multi-dimensionnels. HPF permet des distributions régulières (découpage en blocs ou en cycles) et propose des constructions parallèles assez simples comme les α -opérations (application d'une fonction scalaire sur les éléments des opérandes de même indice) ou des directives d'indépendance d'itérations de boucle (FORALL). La simplicité et la régularité du langage doit permettre une compilation efficace des calculs.

► Irrégularité

Cependant l'analyse numérique ne s'accommodent par toujours de structures régulières (utilisation de matrices creuses par exemple) et le parallélisme ne se limite pas au calcul numérique. Cela a expliqué l'émergence de l'irrégularité dans certains langages (VIENNA FORTRAN, C* [Ros87]...). L'irrégularité peut cependant revêtir différents aspects suivant qu'on s'intéresse aux données ou au code et suivant qu'on considère la structure ou le temps.

En terme de structures de données, VIENNA FORTRAN [UZCZ97] vise essentiellement à étendre les possibilités de distributions : on peut spécifier un découpage en blocs non uniformes et même proposer un mapping complet des tableaux.

D'autres langages vont plus loin et décident de ne plus référencer les éléments

par un indice dans une structure mais comme des unités autonomes (processeurs virtuels) reliées les unes aux autres par une topologie particulière (c'est-à-dire par les liens avec ses voisins). Dans ce contexte, il est plus aisé d'envisager l'extension ou la réduction dynamique des structures de données comme c'est le cas pour COIR [SG95] et IDOLE [Kok96].

Tous les langages admettent l'irrégularité structurelle du code qu'ils ont héritée du mode séquentiel. Par contre, peu nombreux sont ceux qui permettent la dynamicité de l'exécution, elle s'obtient généralement en gérant un drapeau d'activité [KP93b] sur les éléments d'une structure (là encore cette notion est plus naturelle pour les langages utilisant la sémantique du processeur virtuel).

► Expression de l'algorithme

On retrouve évidemment les concepts qui existent déjà en séquentiel (fonctionnel, objet...). On a aussi des concepts bien spécifiques.

Dans le cas du parallélisme de tâches, ces concepts sont la possibilité évidemment d'avoir des exécutions parallèles (dans OCCAM, comme un peu dans ESTEREL, on spécifie ce qui peut se faire en parallèle et ce qui se fait en séquence) et également d'avoir un moyen de synchroniser les exécutions. Cette synchronisation est obtenue par divers moyens qui vont de la simple attente de fin de procédure (RPC) aux communications en passant par les sémaphores.

Suivre un algorithme en séquentiel n'est pas une chose forcément aisée mais, pour les langages « bien » structurés (sans ruptures de séquences), il existe des outils formels d'analyse de code (test d'arrêt, vérification de protocole...). Ce n'est plus le cas en parallélisme. Pourtant si on se limite au parallélisme de données, on peut adopter une structure de contrôle totalement séquentielle et faire intervenir le parallélisme en utilisant des α -opérations ou β -opérations comme c'est le cas dans les langages FORTRAN. On retrouve alors certaines bonnes propriétés des codes séquentiels [Bou91, Bou93].

Cependant le strict respect d'un tel codage entraîne un problème d'efficacité dû aux trop nombreuses synchronisations. Ceci explique que la plupart des langages autorisent des structures conditionnelles dans leur opérations parallèles, ce qui nous ramène aux difficultés de validation et compilation d'algorithme précédemment citées.

► Interactivité

Certains langages, ou plutôt, certaines applications proposent une interface de calculs interactifs (notamment MATLAB [Mat94]). Ceux-ci n'ont évidemment pas l'efficacité requise pour l'exécution d'applications TS *in vivo*. Mais ils peuvent quand même avoir une utilité non négligeable dans un atelier de développement en terme de simulation et de débogage.

1.3 Conclusion et application au TS

Les algorithmes des applications TS sont relativement simples du point de vue du contrôle d'exécution. Leur expression dans n'importe lequel des langages cités ne posent généralement guère de problèmes (mis à part le problème des flots de données infinis qui sont somme toute assez courants) et produit une exécution optimale en mémoire partagée du point de vue algorithmique (sans souci de taille mémoire). En effet, la taille des données à traiter fait que l'on peut estimer qu'ils sont en nombre suffisant pour alimenter tous les processeurs à part égale.

La vraie question concerne donc plutôt la distribution des données et des calculs dans un contexte à mémoire distribuée, et la transformation du programme aussi bien en parallèle qu'en séquentiel pour répondre aux contraintes physiques et dynamiques. On peut estimer, à juste titre, que ces problèmes ne doivent pas apparaître au niveau de la spécification de l'application mais que celle-ci devrait clairement mettre en avant le côté régulier des traitements. Cela a justifié pour TMS la définition d'un langage spécifique, ARRAY-OL, pour la partie transformationnelle des applications de TS.

Chapitre 2

L'environnement de programmation ARRAY-OL et ses alternatives

Le précédent chapitre a délimité l'étendue du domaine du traitement de signal sur lequel nos travaux se positionnent. Il s'agit de la partie transformationnelle des signaux. Il a également mis en avant l'avantage d'utiliser un nouveau langage de description des applications tirant partie des spécificités de ce domaine. C'est pour ces raisons que TMS a créé le langage ARRAY-OL [DLB⁺95] (Array-Oriented Language).

Ce langage se situe à l'intersection des domaines du traitement de signal et du traitement de données. Il est destiné à faciliter non seulement l'écriture des algorithmes pour les utilisateurs mais également leur compilation puisqu'il a été conçu de manière à exprimer les traitements le plus directement possible (sans les traduire dans un formalisme de boucle, par exemple). De plus, sa spécification a été menée en parallèle de celle d'une architecture matérielle qui doit permettre son exécution efficace dans un contexte embarqué.

La description précise du langage et de sa chaîne de compilation font l'objet des deux premières sections du chapitre.

Découlant naturellement de la spécification d'une application dans ce langage, il existe une implémentation évidente dans à peu près n'importe quel langage de traitement numérique. Seulement elle souffre de grandes limitations que nous exposerons dans la troisième section après avoir explicité le schéma de compilation directe dont nous parlons.

Cela nous amènera à envisager un moyen de transformer ces applications pour obvier aux problèmes soulevés. Les deux dernières sections seront consacrées à la description de quelques environnements existants qui visent les mêmes objectifs et des modèles formels sur lesquels ils reposent. Nous expliqueront finalement en quoi ces environnements ne nous satisfont pas et ce que nous proposons.

2.1 Un atelier de développement d'applications TS

Autour du langage ARRAY-OL, TMS a mis en place un atelier complet de développement d'applications TS. Trois pôles peuvent être ainsi identifiés :

- Dans le souci de faciliter la production de programmes ARRAY-OL tout en libérant le programmeur des contraintes syntaxiques liées à l'utilisation d'un langage de programmation, TMS propose une interface de programmation visuelle basée sur l'environnement PTOLEMY.
Elle permet de spécifier graphiquement (programmation par l'exemple) une application ARRAY-OL par l'enchaînement de tâches élémentaires (cf. 2.2.2). Par le biais d'un *compréhenseur*, elle génère de façon automatique les fichiers de code correspondant dans la syntaxe textuelle ARRAY-OL.
- Une fois l'application écrite, il est souhaitable de simuler son exécution sur une ou plusieurs stations de travail afin de la valider. Une proposition pour cette phase forme l'essentiel de nos travaux.
- L'objectif étant d'exécuter de manière efficace ces applications en situation réelle, TMS a conçu une architecture spécialisée (l'accélérateur synchrone, ACC-SYNC). Les applications ARRAY-OL ont vocation à être portées sur cette architecture. En l'état actuel, cela se fait par micro-codage en dur des fonctions dans l'ACC-SYNC, d'où l'intérêt de valider l'algorithme sur station de travail avant son implantation sur la machine de production.

2.2 Le langage ARRAY-OL

La description d'une application en ARRAY-OL fait successivement appel à deux niveaux. Le premier niveau définit l'enchaînement des différentes parties de l'application dans son ensemble alors que le second précise les actions élémentaires à effectuer. Le modèle de fonctionnement sous-jacent déclenche les différentes étapes de calcul en fonction des dépendances exprimées au premier niveau ; chaque étape de calcul s'exécute suivant le modèle de fonctionnement SPMD.

2.2.1 Le modèle global

► Le graphe des tâches

Le modèle global permet de nommer et de définir les tableaux. Ce sont des tableaux multi-dimensionnels non nécessairement finis. Ils sont les seules entités utilisées pour agencer le graphe de dépendances des tâches : chacune des tâches prend ses entrées parmi les tableaux définis et produit des tableaux 2.1. La spécification de la tâche ainsi que le balayage des tableaux ne sont pas visibles à ce niveau.

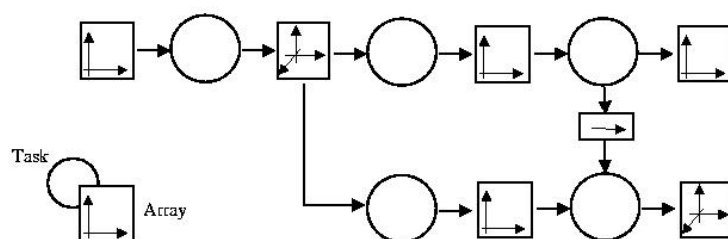


FIG. 2.1 – Graphe de dépendances entre les tâches d'une application ARRAY-OL

► Les tableaux ARRAY-OL

Les applications de TS sont organisées autour d'un flot de données potentiellement infini. ARRAY-OL capture ce flot infini, qui représente généralement le temps, dans des tableaux multi-dimensionnels dont l'une des dimensions peut être infinie. De plus, il est fréquent que certaines dimensions spatiales correspondent à des capteurs qui peuvent, par exemple, être organisés en cercle. Les dimensions finies des tableaux ARRAY-OL sont donc toriques.

► L'exemple de la Veille à Bande Large

Nous allons illustrer la définition du langage que nous sommes en train de donner sur l'exemple de Veille à Bande Large (VBL) dont le schéma est donné sur la figure 2.2. Il s'agit de la première phase de traitement d'une chaîne sonar conventionnelle. Son entrée est constituée par un anneau d'hydrophones produisant un flux continu de données. Ceux-ci sont d'abord traités fréquentiellement puis regroupés en voies ; une voie représente plus ou moins une direction spatiale donnée. Ensuite ces voies sont normalisées puis intégrées, de façon à extraire des concordances de fréquences provenant d'une même direction (donc appartenant vraisemblablement au même objet).

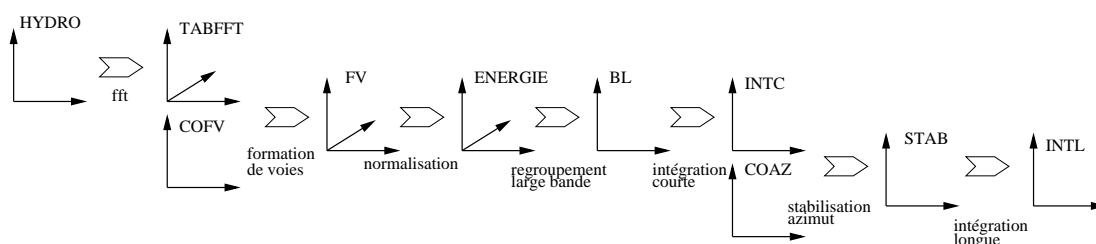


FIG. 2.2 – Schéma de la VBL

La description complète de cette application en ARRAY-OL est donnée en annexe A.1.

2.2.2 Le modèle local

ARRAY-OL bénéficie de la simplicité des opérations du TS. Une tâche ARRAY-OL met en relation des tableaux d'entrée et de sortie. Une tâche est toujours composée d'un constructeur d'itérations, où chaque itération est indépendante et appliquée sur un sous-ensemble fini de points des différents tableaux en entrée et en sortie. Ces sous-ensembles, ou motifs, doivent être réguliers (ajustage du tableau) et se répartir régulièrement sur les tableaux (pavage de tableaux).

La cohérence globale d'une tâche est assurée par le fait que les motifs des tableaux qui sont attachés à une même tâche sont en nombre identique et reliés un pour un entre chacun des tableaux. Ainsi le rôle d'une tâche consiste, pour chaque itération de pavage, à extraire les motifs d'entrée, à leur appliquer une fonction donnée qui produit les valeurs associées aux motifs en sortie, et à finalement ranger ces derniers dans les tableaux résultats.

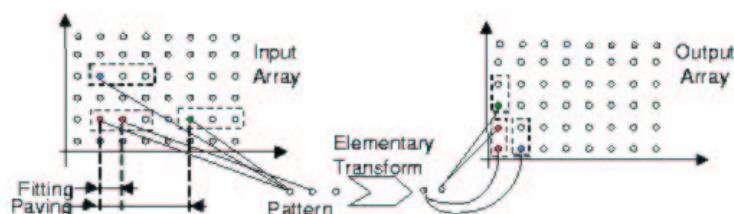


FIG. 2.3 – Exemple de trois itérations d'une tâche. Chaque motifs en entrées (3 points horizontaux) correspond à un motif de sortie (2 points verticaux)

Une valeur d'ajustage et de pavage est spécifiée pour chaque tableau et chaque tâche où apparaît celui-ci. Un même tableau peut donc être pavé/ajusté différemment en fonction de la tâche qui l'utilise (ce qui est particulièrement vrai pour les tableaux qui sont produits puis consommés).

► Ajustage d'un tableau par une tâche

On décrit maintenant un ajustage donné. Les motifs construits sont des tableaux ayant tous les mêmes dimensions. Ils sont définis par la donnée de leur nombre de dimensions (qui peut être différent du tableau d'origine) et pour chaque dimension de sa taille, d'un vecteur du tableau (vecteur d'ajustage).

Les motifs se distinguent par leur point d'origine dans le tableau. À partir de celui-ci, les autres points s'obtiennent par un déplacement le long des vecteurs d'ajustage autant de fois que le nécessite la taille du motif.

Si on reprend l'exemple de la figure 2.3, les motifs n'ont qu'une dimension. Le motif opérande est de taille 3 et défini par le vecteur d'ajustage $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, alors que celui résultat

est de taille 2 et défini par le vecteur d'ajustage $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

► Pavage des tableaux par une tâche

Autant les ajustages sont indépendants entre eux, autant les pavages des tableaux d'une même tâche sont corrélés : ils expriment le lien entre ce qui est utilisé et ce qui est produit.

Le pavage décrit comment obtenir les origines des motifs des tableaux et exprime les liens entre les motifs des différent tableaux. Il est spécifié par la donnée, pour chaque tableau, d'un point d'origine globale et d'un ensemble de vecteurs (vecteurs de pavage) dont chacun est associé à une borne d'itération maximale. Les points d'origine des motifs sont donc obtenus par des déplacements successifs le long des vecteurs de pavage le nombre de fois fixé par les bornes d'itérations et ce, à partir du point d'origine global donné.

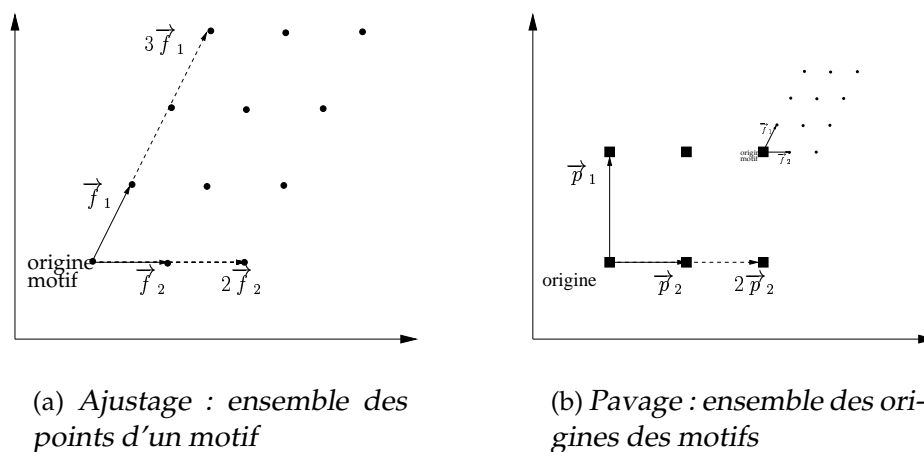


FIG. 2.4 – Pavage et ajustage des motifs :

L'ensemble des points d'un motif (figure de gauche) sont produits par l'itération des 2 vecteurs d'ajustage (\vec{f}_1 , \vec{f}_2) itérés respectivement 3 et 2 fois. Les bornes sont donc 4 et 3 et le motif a 12 points.

L'ensemble des origines des motifs (figure de droite) sont produites par l'itération des 2 vecteurs de pavage (\vec{p}_1 , \vec{p}_2) itérés respectivement 1 et 2 fois. Les bornes sont donc 2 et 3 et il y a 6 motifs.

La cohérence de tâche dont nous avons parlé impose que le nombre de vecteurs de pavage des tableaux d'une même tâche soient identiques ainsi que les bornes d'itérations pour ces vecteurs. Ces valeurs d'itération sont fonctions des dimensions d'un

tableau résultat particulier (appelé *maître*) puisqu'elles doivent permettre de le recouvrir entièrement par les motifs ainsi construits et que, par définition, il ne peut y avoir de recouvrement entre deux motifs d'un tableau de sortie.

► Exemple sur la VBL

Considérons les deux premières tâches de la VBL : FFT et FORMATION_DE_VOIES

– Pour la FFT,

- le tableau d'entrée, `Hydro`, a 2 dimensions ($temps \times hydro$) = $(\infty \times 512)$; le tableau de sortie, `TABFFT`, a 3 dimensions ($temps \times hydro \times freq$) = $(\infty \times 512 \times 256)$
- les motifs d'entrée sont des tranches temporelles contiguës de 512 unités sur chaque hydrophone.

Le vecteur d'ajustage est $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ et le vecteur d'itération, (512) .

Les vecteurs de pavage sont $\begin{pmatrix} 512 & 0 \\ 0 & 1 \end{pmatrix}$ et le vecteur d'itération, $\begin{pmatrix} \infty \\ 512 \end{pmatrix}$

- les motifs de sortie prennent une dimension fréquentielle entière.

Le vecteur d'ajustage est $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et le vecteur d'itération, (256) .

Les vecteurs de pavage sont $\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ et le vecteur d'itération, $\begin{pmatrix} \infty \\ 512 \end{pmatrix}$

– Pour la FORMATION_DE_VOIES,

- les tableaux d'entrée sont `TABFFT` et `COFV`. Ce dernier a 3 dimensions ($voies \times freq \times hydro$) = $(128 \times 200 \times 192)$; le tableau de sortie, `FV`, a 3 dimensions ($temps \times voie \times freq$) = $(\infty \times 128 \times 200)$
- les motifs d'entrée du tableau `TABFFT` sont des tranches temporelles glissantes de 192 unités sur chaque hydrophone et on ne prend que 200 fréquences sur les 256 (entre 28 et 227).

Le vecteur d'ajustage est $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ et le vecteur d'itération, (192) .

Les vecteurs de pavage sont $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et le vecteur d'itération, $\begin{pmatrix} \infty \\ 128 \\ 200 \end{pmatrix}$

L'origine du pavage est $\begin{pmatrix} 0 \\ 0 \\ 28 \end{pmatrix}$

- les motifs d'entrée du tableau `COEFV` sont des tranches de 192 unités s'appliquant sur les hydrophones. Ces coefficients varient avec les hydrophones, les

voies et les fréquences mais pas avec les itérations temporelles.

Le vecteur d'ajustage est $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et le vecteur d'itération, (192).

Les vecteurs de pavage sont $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ et le vecteur d'itération, $\begin{pmatrix} \infty \\ 128 \\ 200 \end{pmatrix}$

– les motifs de sortie sont réduits à un seul point représentant une voie.

Il n'y a donc pas d'ajustage.

Les vecteurs de pavage sont $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et le vecteurs d'itération, $\begin{pmatrix} \infty \\ 128 \\ 200 \end{pmatrix}$

► Le modèle « ARRAY-OL exemple »

Le modèle tel que nous l'avons décrit présente un aspect plutôt algébrique (on parle alors de matrice de pavage et d'ajustage, cf. 2.2.6). Pour offrir aux utilisateurs, concepteurs d'applications TS, un formalisme plus intuitif, TMS a développé le concept de l'exemple pour décrire le pavage et l'ajustage.

On ne rentre plus les vecteurs tels quels mais on donne le point d'origine et le point suivant sur chacune des dimensions, et ceci, aussi bien pour l'ajustage que pour le pavage. Ensuite un programme, le *compréhenseur*, en déduit les bornes d'itérations : pour l'ajustage, il se fie aux dimensions des tableaux arguments de la TE ; pour le pavage, il les calcule à partir des motifs du tableau maître en faisant en sorte qu'ils respectent les contraintes de non-recouvrement et de couverture complète (cf. 2.2.5).

2.2.3 Action élémentaire d'une tâche

► La bibliothèque de composants

Les transformations élémentaires (TE) à effectuer sur les motifs font partie des opérations classiques du TS (FFT, intégration...). On supposera pour la suite qu'elles sont regroupées dans une bibliothèque annexe. Elles constituent autant de *boîtes noires* dont le prototype est fixé en dur : elles utilisent les motifs d'entrée pour remplir les motifs de sortie. Cependant une TE peut être paramétrée, par exemple, par la taille des tableaux qu'elle va traiter.

2.2.4 La hiérarchie

Comme tout langage évolué, ARRAY-OL supporte la notion de modularisation c'est-à-dire la possibilité de réutiliser un « morceau de source » dans d'autres applications.

Le « morceau » en question étant un graphe de tâches, il est plus que probable qu'il contienne des tableaux qui ne sont produits par aucune tâches et d'autres qui ne sont utilisés par aucune. Ces deux ensembles de tableaux peuvent être considérés comme respectivement l'ensemble des arguments et l'ensemble des tableaux de retour de cette sous-application. Il est également envisageable et même souhaitable que cette sous-application accepte des paramètres comme par exemple la taille de certains tableaux qui seront fixés par l'appelant lors de l'utilisation.

En ce qui concerne l'appelant, la première méthode qui vient à l'esprit est naturellement celle qui consiste à insérer la sous-application dans le graphe de tâches. Le passage des arguments se font alors par tableaux entiers. Nous appellerons cette notion le **regroupement**.

On peut remarquer que le **regroupement** est indépendant des spécificité du langage (ajustage/pavage). Ceci nous amène à considérer une autre notion de sous-application, la **hiérarchisation**. En nous basant sur la spécification du langage ARRAY-OL qui impose que les motifs d'une tableau sont de la même forme et que l'action élémentaire d'une tâche consiste à prendre un motif de chaque tableau d'entrée pour calculer un motif résultat de chaque tableau de sortie, on peut alors insérer l'appel de la sous-application au niveau du calcul des motifs. Dans ce cas les tableaux d'entrée et de sortie de cette dernière seront les motifs de la tâche appelante.

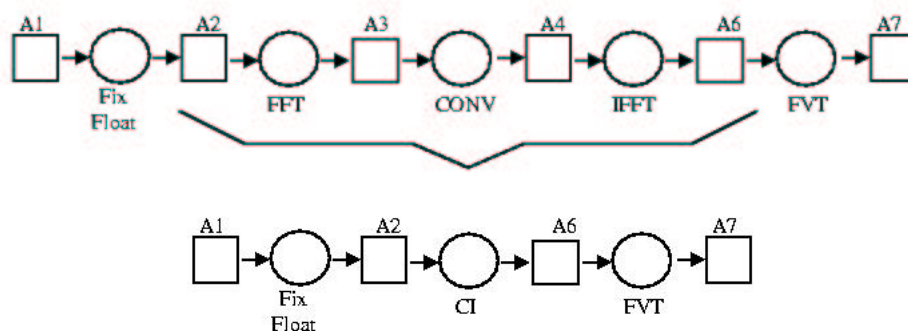


FIG. 2.5 – Hiérarchisation d'une application ARRAY-OL par construction d'une sous-application C1

On peut noter que la sous-application prend alors la place d'une transformation élémentaire pour la tâche. Une fois désigné rien ne distingue donc de l'extérieur une tâche dont l'action est une TE d'une tâche hiérarchique. On peut citer en exemple la présence d'une TE (codé en C++ pour la simulation et micro-codé pour l'ACC-SYNC) permettant de calculer un FFT à 2 dimensions ainsi que la spécification en ARRAY-OL d'une application effectuant le même calcul. L'une et l'autre pouvant s'utiliser de manière équivalente et transparente.

Cette notion de hiérarchie a également une grande importance dans la conception des applications car elle permet un découpage par raffinement des algorithmes

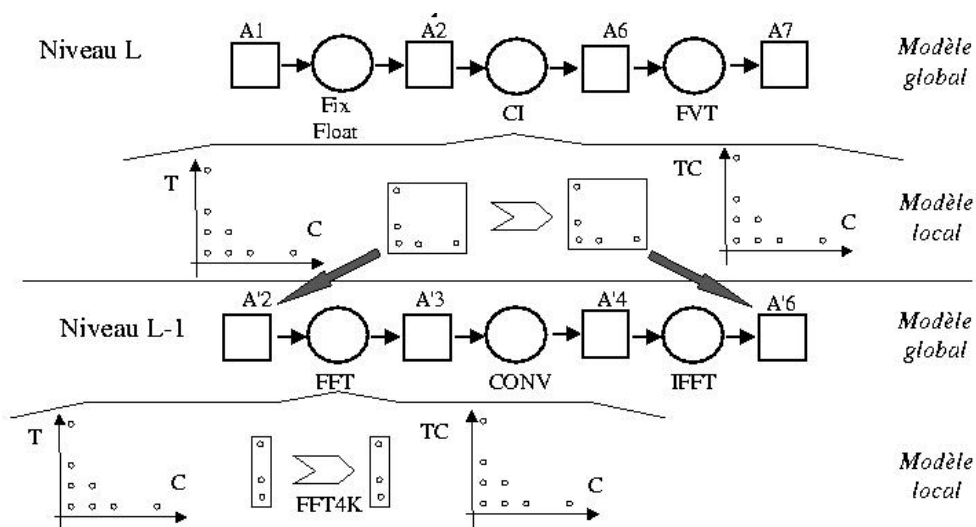


FIG. 2.6 – Forme hiérarchique détaillé de la sous-application C1

de calcul en vu de leur placement sur une architecture donnée. Ainsi l'architecture de l'ACC-SYNC est formée d'une hiérarchie à quatre niveaux :

- un certain nombre de nœuds forment le premier niveau ;
- chacun des nœuds se réplique dans le temps en déroulant un programme ;
- un nœud est formé d'un ensemble de processeurs élémentaires, PE ;
- chaque PE se réplique dans le temps en déroulant un programme.

2.2.5 Contraintes du langage

Les spécifications d'ARRAY-OL incluent un certain nombre de contraintes sur le remplissage des tableaux résultats :

- tout point d'un tableau de sortie est calculé une et une seule fois. Les motifs de sortie ne se chevauchent pas et pavent entièrement le tableau ;
- les vecteurs d'ajustage et de pavage associés aux tableaux résultats sont parallèles aux axes ;
- les éléments des vecteurs d'ajustage et de pavage résultats sont positifs ;
- les tableaux résultats ne sont pas toriques : on ne doit pas utiliser le modulo lors du déplacement le long des vecteurs de pavage et d'ajustage.

D'autres contraintes concernent le modèle global :

- on impose que les dimensions des tableaux (et, *a fortiori*, les bornes de pavage des tâches) ne comportent, au plus, qu'une dimension infinie ; cela représente sémantiquement un flux continu de données ;
- les motifs sont forcément de taille finie ;
- le graphe des tâches d'une application ne comporte pas de cycle ;
- dans le cas d'une application hiérarchique, un tableau d'entrée ou de sortie d'une

sous-application (qui est aussi un motif de l'application supérieure) n'est pas considéré comme torique au niveau de la sous-application, il n'y a donc pas de calcul de modulo à ce niveau.

En fait, seule la première contrainte (sur l'unicité des points des tableaux résultats) est indispensable à une sémantique déterministe d'ARRAY-OL, conséquence du calcul en parallèle des motifs. Les autres sont des simplifications *raisonnables* compte tenu du domaine d'application qu'est le traitement de signal.

2.2.6 Le formalisme matriciel

La spécification ARRAY-OL peut se représenter par une description matricielle. Nous introduisons ici cette représentation.

► Définitions

Soit T , une tâche, alors \mathcal{A}_T^{in} , \mathcal{A}_T^{out} et \mathcal{A}_T désignent respectivement les tableaux d'entrée, de sortie et l'ensemble des tableaux de la tâche.

Soit $M \in \mathcal{A}_T$ alors

- $\mathcal{P}_{M,T}$ (resp. $\mathcal{F}_{M,T}$) désigne la matrice constituée de l'ensemble des vecteurs de pavage (resp. d'ajustage) ;
- Q_T et $D_{M,T}$ sont des vecteurs désignant respectivement les bornes d'itération du pavage¹ et les dimensions du motif ;
- enfin $\mathcal{O}_{M,T}$ définit l'origine du pavage dans le tableau.

► Quelques caractéristiques

On a autant de lignes dans $\mathcal{P}_{M,T}$, $\mathcal{F}_{M,T}$ et $\mathcal{O}_{M,T}$ que de dimensions au tableau. Le nombre de colonnes des matrices de pavage et d'ajustage sont respectivement égale à la taille des vecteurs $Q_{M,T}$ et $D_{M,T}$.

$\{\text{mod}_M(\mathcal{P}_{M,T} \cdot \chi_q + \mathcal{O}_{M,T}); \forall \chi_q, 0 \leq \chi_q < Q_T\}$ représente donc les coordonnées de l'ensemble des origines des motifs et $\{\text{mod}_M(\mathcal{P}_{M,T} \cdot \chi_q + \mathcal{O}_{M,T} + \mathcal{F}_{M,T} \cdot \chi_d); \forall \chi_d, 0 \leq \chi_d < D_{M,T}\}$ représente l'ensemble des points du motif pour l'itération χ_q .

► Un exemple

Dans le cas de la deuxième tâche de la VBL, `FORMATION_DE_VOIES`, on a :

¹rangées dans le même ordre évidemment que les vecteurs dans la matrice.

$$\begin{aligned}
 A_T^{in} &= \{\text{Fft}, \text{Coefv}\} \\
 A_T^{out} &= \{\text{Fv}\} \\
 Q_T &= \begin{pmatrix} \infty \\ 512 \\ 200 \end{pmatrix} \\
 D_{\text{Fft},T} &= (192)
 \end{aligned}
 \quad
 \begin{aligned}
 \mathcal{P}_{\text{Fft},T} &= \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{vmatrix} \\
 \mathcal{F}_{\text{Fft},T} &= \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix}
 \end{aligned}$$

2.2.7 Représentation textuelle

Bien que ARRAY-OL soit, à la base, un langage de description visuelle, une forme écrite est indispensable pour pouvoir échanger et traiter des applications.

► Format AOL

Une première forme textuelle, d'extension `.aol`, a été créée par TMS. Les objets sont décrits par une succession de déclarations `clé {valeur}`. Les valeurs peuvent être un type terminal (entier, texte, liste...) ou une structure elle-même constituée d'une succession de déclarations `clé {valeur}`.

- La structure de premier niveau est `application`. Elle contient quelques informations pratiques ainsi que la définition des tableaux et des tâches

```

application {
  name {VBL}
  version { @(#)VBL.aol 1.4 5/9/96 TSASM/SMI }
  author { D. Rozzonelli }
  array { ... }
  ...
  task { ... }
  ...
}

```

- les clés `array`, `array in` et `array out` définissent le nom et les caractéristiques d'un tableau. Le suffixe précise si celui-ci est interne, en entrée ou en sortie de l'application (le symbole `~` représente l'infini)

```

array {
  name           { TABFFT }
  dimension      { 3 }
  names          { (h, f, ra) }
  origins        { (0, 0, 0) }
  sizes          { (512, 256, ~) }
  type           { Complex }
}

```

- la clé `task` définit le nom et les caractéristiques d'une tâche

```
task {
  name { FORMATION_DE_VOIES }

  use { DotCx(192) }

  slave in { ... }
  slave in { ... }
  master out { ... }
}
```

- la clé `use` donne le nom de la TE
- les clés `slave in` et `slave out` définissent le nom des tableaux opérands (`in`) ou résultats (`out`), le nom des motifs et les vecteurs de pavage et d'ajustage

```
slave in {
  name { TABFFT }
  pattern { IN0 }
  origin { (0, 28, 0) }
  fitting { ((1, 0, 0)) }
  paving { ((4, 0, 0), (0, 1, 0), (0, 0, 1)) }
}
```

- la clé `master out` définit le tableau résultat maître et ses caractéristiques. Elles sont du même type que ci-dessous mais avec en plus le vecteur d'itération du pavage (`qd`)

```
master out {
  name { FV }
  pattern { OUT0 }
  origin { (0, 0, 0) }
  fitting { ( ) }
  paving { ((1, 0, 0), (0, 1, 0), (0, 0, 1)) }
  qd { (128, 200, ~) }
}
```

► Format TAOL

TMS a depuis fait évoluer cette forme vers une représentation plus compacte et surtout faisant plus clairement apparaître la distinction entre le niveau global et le niveau local. Ceci a abouti au format d'extension `.taol`.

- la première partie ne sert qu'à donner des noms aux objets (application, tableaux, motifs et tâches) et à définir le graphe de tâches :

```
nom_de_l'application < tâche ; tâche ...
```

La spécification des tâches est constitué du nom de la TE encadré par le nom des motifs d'entrée et de sortie, et, le tout, encadré des noms des tableaux opérandes et résultats avec leur types (le point d'exclamation désigne le tableau maître) :

```
tab1 tab2 [ ( motif1 motif2 ) TE ( motif3! ) ] tab3 ;
```

En reprenant l'exemple de la VBL :

```
Main< Hydro [( IN_IN0 ) Fft ( OUT_OUT0! )] Tabfft{complex<double>} ;
      Tabfft{complex<double>} Cofv{complex<double>}
          [( IN_IN0 IN_IN1 )
           F_voies
           ( OUT_OUT0! )]
          Fv{complex<double>} ;
      ....
  >
```

- la suite est constituée par la définition des caractéristiques des objets (tableau, pavage, ajustage...). Les objets sont désignés par la hiérarchie des noms (Main.Tabfft ou Main.F_voies.IN_IN0).

```
// tableau
Main.Hydro ( 512 ~ )
// parametres TE
Main.F_voies : DotCx(IN_IN0:IN0, IN_IN1:IN1, OUT_OUT0:OUT0)
// motif
Main.F_voies.IN_IN0 ( 192 )
// attribues de tache
Main.F_voies/$Q = ( 128 200 ~ )
Main.F_voies.IN_IN0/$A = ( 1 ; 0 ; 0 ; )
Main.F_voies.IN_IN0/$P = ( 4 0 0 ; 0 1 0 ; 0 0 1 ; )
Main.F_voies.IN_IN0/$O = ( 0 28 0 )
```

► Format de commande GASPARD

Nous avons également défini, dans notre environnement de programmation, une forme textuelle d'ARRAY-OL. Il s'agit en fait d'un format d'interrogations et de manipulations de structures ARRAY-OL, il sert d'interface entre le moteur de l'environnement

et les autres composants (comme l'interface graphique). Nous nous en servons comme format de sauvegarde en considérant qu'une application est définie par la succession des commandes qui permettent de la recréer. Ce format est succinctement décrit dans la partie traitant de l'environnement (chapitre 4).

2.3 Compilation d'un code ARRAY-OL

ARRAY-OL est un langage de spécification, il permet de décrire l'algorithme de calcul des tableaux mais ne propose aucune directive pour expliciter la façon de l'exécuter. Au niveau global, les dépendances entre les tâches sont données par les tableaux d'entrée et de sortie. Au niveau local, les dépendances sont explicitées par l'unité des pavages des différents tableaux d'une tâche.

Un compilateur se doit de respecter ces dépendances entre données. Il lui faut aussi faire un certain nombre de choix : l'allocation et le placement des tableaux doivent être définis ; l'ordre dans lequel seront effectués les calculs tant au niveau global (enchaînement des tâches) qu'au niveau local (boucles SPMD indépendantes) doit être précisé. Cependant, les choix faits par le compilateur sont déterminant dans l'obtention de performances.

Cette liberté dans l'exécution devrait nous permettre de générer du code efficace en fonction de la machine visée :

- l'exécution sur la machine dédiée ACC-SYNC tire partie d'un placement optimisé sur les différents niveaux de hiérarchie de la mémoire. Le placement garantit la minimisation des communications entre les niveaux tout en respectant les tailles mémoires ;
- la mise en œuvre d'un ordonnancement particulier doit permettre de diminuer l'espace mémoire nécessaire à l'exécution sur une station de travail Unix. Sur un réseau de stations, il devrait assurer une bonne répartition des données entre les processeurs.

Notre approche est de décrire dans une première étape un compilateur qui assure le respect de la sémantique des applications ARRAY-OL. Nous détaillons quelques points délicats et sensibles de cette compilation. Ce compilateur ne pourra prendre en compte l'ensemble des applications ARRAY-OL. En particulier, la gestion des tableaux infinis sera restreinte. Nous proposerons ensuite des transformations de code ARRAY-OL en code ARRAY-OL pouvant être compilé efficacement par ce compilateur basique. Ce travail de transformation de code est le cœur de nos travaux.

2.3.1 Schéma d'implantation séquentiel direct

Au niveau global, l'absence de cycle permet de trouver un ordonnancement linéaire des tâches respectant les dépendances. Il y a donc des tableaux qui ne sont produits par

aucune tâche et d'autres qui ne sont utilisés par aucune tâche. Si on implémente une application comme une fonction, ces tableaux constitueront les arguments d'entrée et de sortie. On peut considérer que l'application n'a pas à se soucier de ceux-ci, ils sont alloués et gérés à l'extérieur (par le composant d'acquisition, par exemple). L'application ne prend en charge que les tableaux strictement internes.

L'exécution d'une application se résume à l'exécution séquentiel de ses tâches. Chaque tâche est, de par la nature du langage, constituée naturellement d'un nid de boucles représentant les itérations de pavage (unique pour une tâche). Au sein d'une itération, on doit construire les motifs d'entrée (i.e. des tableaux d'entrée), appeler la fonction élémentaire et reporter les motifs de sortie dans leur tableau respectif. La figure 2.7 schématise un tel code.

```
application1(A,B) {...}
application2(A,B,C) {...}
application3(A,B) {

  /* tache 1 */
  forall ()
    forall () ...
      remplir_motif ()
      appel_transformation_elementaire ()
      utiliser_motif ()

  /* tache 2 */
  ....
}
...
```

FIG. 2.7 – Structure générale de l'exécution d'une application ARRAY-OL

À noter que le choix fait pour implémenter une application est cohérent avec l'appel des fonctions élémentaires, ce qui signifie qu'on n'a pas à distinguer l'appel d'une transformation élémentaire et l'appel à une sous-application.

Du point de vue de la structuration interne, l'application principale (point d'entrée du programme) ne se distingue pas des autres sous-applications. La seule différence tient au fait que : premièrement, ses tableaux d'entrée et de sortie proviennent de l'extérieur (du système de fichiers sur station de travail, ou des composants d'entrées/sorties pour la machine dédiée); deuxièmement, elle est la seule à autoriser des tableaux de taille infinie.

2.3.2 Points sensibles de la génération de code

Considérant que les transformations élémentaires sont fournies au compilateur sous la forme de boîtes noires, la partie prédominante de l'exécution sur laquelle nous pouvons influencer est constituée par la gestion des tableaux et des motifs. La suite expose différents points critiques de cette partie et pour certains d'entre eux une manière d'améliorer leur efficacité.

► Parcours des tableaux

La spécification d'une application ARRAY-OL ne donne pas d'ordre de parcours des dimensions des tableaux. Cet ordre doit donc être choisi lors de la compilation afin de favoriser les accès consécutifs dans la même page de la mémoire virtuelle et d'éviter le phénomène de *va-et-vient* du swap (cf. figure 2.8).

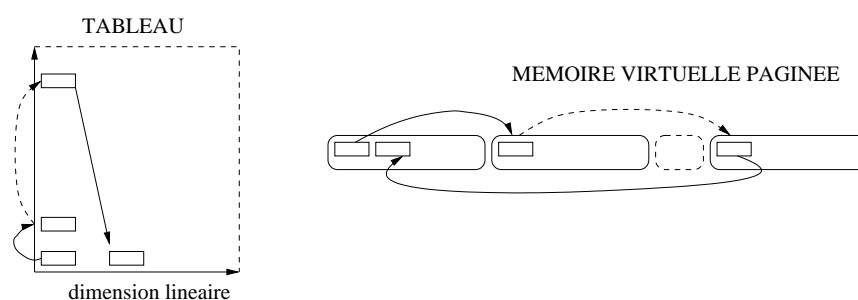


FIG. 2.8 – Illustration du phénomène de *va-et-vient*

En ce qui concerne les tableaux d'entrée et de sortie (qui sont livrés en l'état) on ne peut jouer que sur l'ordre des dimensions du vecteur d'itération. Par contre, les tableaux internes étant gérés exclusivement par l'application, on peut aussi influencer lors de leur création sur l'ordre de linéarisation des dimensions.

Cependant les tableaux sont généralement pavés/ajustés de plusieurs manières (lors de leur production et lors de leur consommation, par exemple). Ces différents modes d'accès peuvent être radicalement opposés, par exemple un tableau 2-D produit par ligne et consommé par colonnes. Cette caractéristique, appelée *corner-turn*, est malheureusement fréquente en traitement de signal. Dans un tel cas, on ne peut définir une allocation satisfaisante pour tous les accès. Cela signifie qu'il faut alors choisir quelle tâche favoriser. Nous laissons ce problème en suspend pour l'instant, il sera traité dans la partie suivante.

► Court-circuit

Dans les spécifications ARRAY-OL, rien ne distingue l'appel d'une transformation élémentaire de l'appel d'une sous-application hiérarchique. Donc, le strict respect du langage devrait nous amener à former les motifs d'entrée/sortie passés en paramètres des sous-applications. Cependant cette copie est coûteuse et surtout inutile si on considère que la sous-application va à nouveau copier les valeurs de ces tableaux dans les motifs de ses propres tâches. Une amélioration consiste à court-circuiter ces recopies (figure 2.9).

Ainsi les sous-motifs (ceux formés par la sous-application) peuvent très bien traiter directement avec les tableaux d'origine : la sous-application reçoit des références sur les tableaux d'entrée et de sortie ainsi que les origines dans ces tableaux du motif courant. Un sous-motif est construit avec l'origine reçue en argument. Les vecteurs de pavage/ajustage sont obtenus par composition des vecteur pavage/ajustage de la sous-application avec ceux de l'application supérieure :

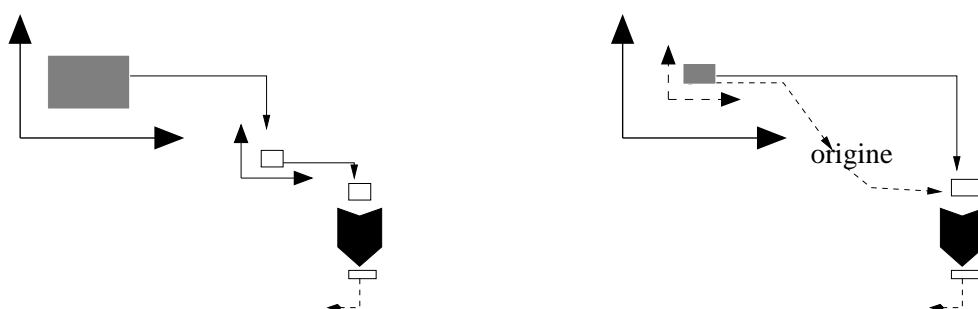


FIG. 2.9 – Illustration du court-circuit des motifs.

- soient \mathcal{P} et \mathcal{F} les matrices de pavage/ajustage de la tâche de l'application supérieure, Q et D leur bornes d'itération et \mathcal{O} l'origine ;
- soient \mathcal{P}' et \mathcal{F}' les matrices de pavage/ajustage d'une des tâches d'entrée ou de sortie de la sous-application, Q' et D' leur bornes d'itération et \mathcal{O}' l'origine ;
- on suppose, dans un premier temps, que la sous-application n'utilise pas de modulo ;
- pour tout point de la sous-tâche désigné par (χ'_q, χ'_d) et appelé dans l'itération χ_q de la tâche supérieure. Le vecteur $\mathcal{O} + |\mathcal{P}'\mathcal{F}'| \begin{pmatrix} \chi'_q \\ \chi'_d \end{pmatrix}$ donne les coordonnées du point dans le système du tableau de la sous-application c'est à dire du motif de la tâche

supérieure. Donc les coordonnées pour le tableau de l'application supérieure sont

$$\begin{aligned} \mathcal{O} + |\mathcal{P}\mathcal{F}| \left(\begin{array}{c} \chi_q \\ \mathcal{O}' + |\mathcal{P}'\mathcal{F}'| \left(\begin{array}{c} \chi'_q \\ \chi'_d \end{array} \right) \end{array} \right) \\ = \mathcal{O} + |\mathcal{F}| \cdot (\mathcal{O}') + \mathcal{P} \cdot \chi_q + |(\mathcal{F} \cdot \mathcal{P}')(\mathcal{F} \cdot \mathcal{F}')| \left(\begin{array}{c} \chi'_q \\ \chi'_d \end{array} \right) \end{aligned}$$

- ainsi les matrices de pavage/ajustage permettant à la sous-tâche d'atteindre directement le tableau sont : $|\mathcal{F} \cdot \mathcal{P}'|$ et $|\mathcal{F} \cdot \mathcal{F}'|$. La partie $(\mathcal{O} + |\mathcal{F}| \cdot (\mathcal{O}') + \mathcal{P} \cdot \chi_q)$ représente l'origine virtuelle du tableau vu par la sous-tâche à cette itération.

Notons que le résultat de cette composition est constant et peut être calculé à la compilation. Par contre, la mise en place de cette stratégie amène à différencier les TE et les sous-applications puisque ces dernières bénéficieront d'un argument en plus qui est le décalage virtuelle du tableau dépendant de l'itération de pavage de la tâche appelante.

Dans certains cas, la transformation proposée pourrait ne pas être possible :

- si la sous-tâche utilisait le modulo ; ce modulo s'insérerait entre la matrice \mathcal{F} et les matrices de la sous-tâche, empêchant les multiplications matricielles. Cependant, d'après les spécifications d'ARRAY-OL hiérarchique, l'utilisation du modulo n'est pas possible dans ce cas ;
- si la sous-application était utilisée par plusieurs applications ; on ne pourrait plus calculer statiquement les matrices finales. Cependant, d'après les spécifications d'ARRAY-OL, une sous-application est spécifique à son application supérieure. Du reste, même en relâchant cette contrainte, il suffit de particulariser la sous-application pour cet appel en la clonant.

2.3.3 Court-circuit vs. pavage incrémental

Quand l'entrée d'une tâche est infini, cela signifie souvent généralement qu'elle représente un flux continu de données. Si c'est le cas, ceci implique qu'on ne peut qu'avancer dans le flux (il ne doit pas être permis de revenir en arrière comme pour un fichier). Pour satisfaire cette contrainte dans ARRAY-OL, il faut que les motifs successifs progressent constamment sur la dimension infinie : les vecteurs de pavage doivent être rangés dans l'ordre croissant et chacun d'eux doit être supérieur à la borne englobante des motifs précédent

$$\mathcal{P}_i \geq \max \left\{ \begin{array}{l} |\mathcal{P}_{[1..i]} \mathcal{F}| \cdot \left(\begin{array}{c} \chi_q \\ \chi_d \end{array} \right) \\ \forall \chi_q, \chi_d \ ; \ \vec{0} \leq \chi_q < Q_{[1..i]}, \vec{0} \leq \chi_d < D \end{array} \right\}$$

Si la tâche en question est en court-circuit avec sa hiérarchie alors les vecteurs de pavage de la sous-tâche d'entrée doivent aussi vérifier les mêmes propriétés. En cas de

non respect, les ensembles de motifs non croissants doivent être recopiés en mémoire pour autoriser les retours en arrière, ce qui correspond à annuler le court-circuit. Pour la tâche supérieure, il faut faire passer les vecteurs de pavage dans l'ajustage pour les mêmes raisons i.e. créer une hiérarchie.

► Accès aux points du motif

En ce qui concerne les motifs que nous sommes obligés de traiter directement sans pouvoir les « court-circuités », le schéma de calcul des points dérivant du langage est assez simple :

$$\text{mod}_{\text{dim}} (\mathcal{P} \cdot \chi_q + \mathcal{F} \cdot \chi_d + \mathcal{O}),$$

Cela signifie que chaque point nécessite deux multiplications matricielles suivies d'un modulo et d'une conversion de vecteur en adressage linéaire. Évidemment la multiplication matricielle découlant du pavage n'est effectuée qu'une seule fois pour un motif donné.

Si on implémente séquentiellement un balayage naturel des itérations c'est-à-dire en ordonnant les dimensions et en choisissant une progression linéaire, les multiplications matricielles peuvent alors être remplacées par des sommes fixes. En effet, chaque incrémentation d'une dimension de l'indice correspond à l'ajout d'un vecteur constant au point précédent. Pour la dernière dimension, il s'agit du vecteur de la matrice, pour les autres il faut également retirer toutes les incrémentations faites sur les dimensions précédentes.

Soient $\mathcal{P} = |\mathcal{P}_1 \dots \mathcal{P}_r|$ les vecteurs de pavage, $\mathcal{F} = |\mathcal{F}_1 \dots \mathcal{F}_s|$ les vecteurs d'ajustage, $\chi_q = (q_1 \dots q_r)$ les indices de pavage du motif courant et $D = (D_1 \dots D_s)$ les bornes d'indice de l'ajustage. Il s'agit alors de calculer statiquement (lors de la génération du code) les vecteurs d'incrémentations de pavage :

$$\mathcal{F}'_i = \mathcal{F}_i - \sum_{j>i} D_j \times \mathcal{F}_j$$

Il reste à calculer, à chaque incrémentation du vecteur, quelle dimension change linéairement (les dimensions inférieures étant remise à 0) et à remplacer l'unique boucle de parcours par un nid de boucles tel que présenté à la figure 2.10.

► Réduction des modules

En aval du calcul des coordonnées de chaque point d'un tableau, une opération de modulo assure que le point appartient au tableau (les tableaux ARRAY-OL sont toriques sur chacune de leurs dimensions). Pour la majorité des motifs, aucun de leurs points ne déborde du tableau. Fréquemment, aucun des points référencés dans un tableau ne nécessite l'utilisation du modulo.

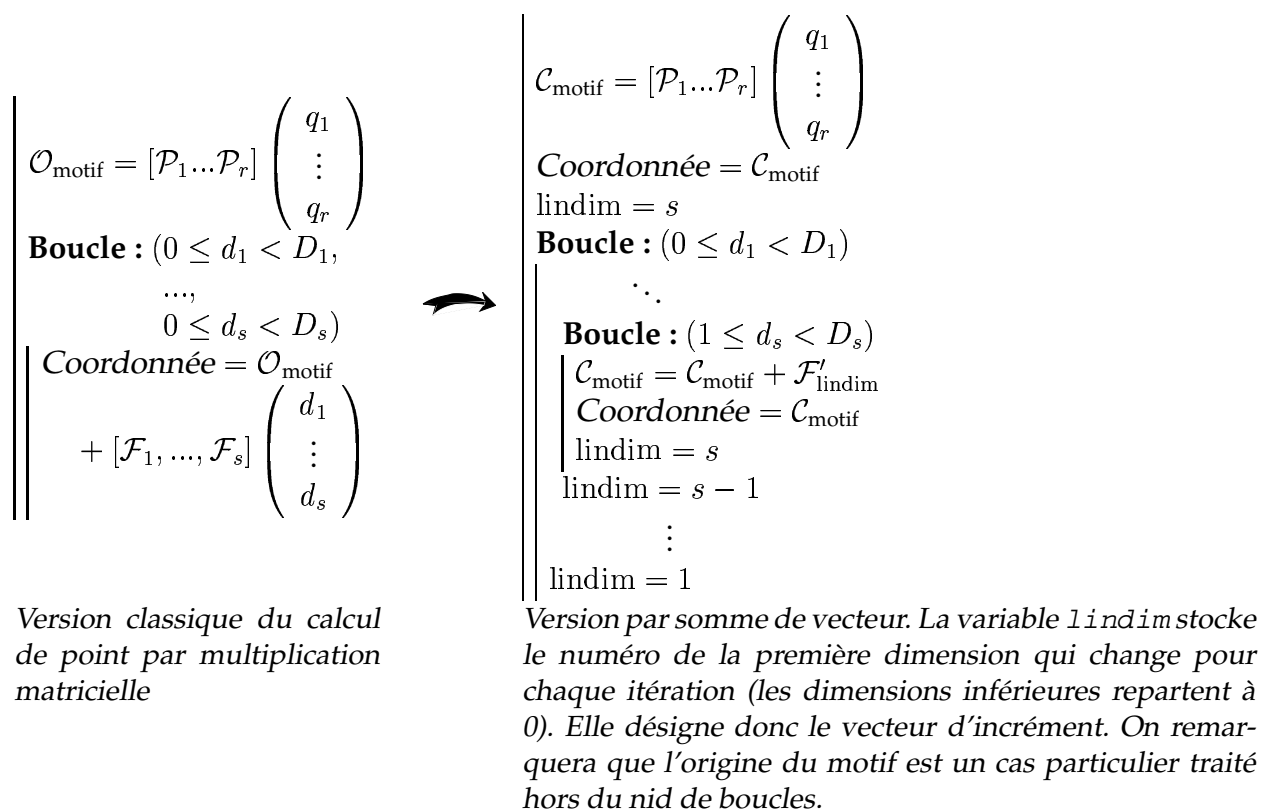


FIG. 2.10 – Réécriture d'une boucle d'accès aux points d'un motif en un nid de boucles

Un calcul simple permet d'identifier préalablement si un ensemble de points obtenus par itérations cartésiennes de vecteurs déborde d'un tableau. La matrice formée par les vecteurs est séparée en deux parties : celle des valeurs positives et celle des valeurs négatives. On effectue le calcul du point pour l'itération maximale sur chacune des deux parties. Les deux vecteurs résultats correspondent respectivement aux bornes minimales et maximales de variation des indices sur chacune des dimensions. Une comparaison de ces bornes avec l'origine et la taille du tableau permet d'identifier si l'utilisation du modulo est nécessaire. Ce calcul est linéaire avec le nombre de dimensions. À noter également que le test est exact : on est assuré qu'à un moment donné, on trouvera un point résultat atteignant n'importe laquelle des limites calculées².

Le compilateur peut dans un premier temps tester l'utilité du modulo pour tous les accès à un tableau donné ; on teste le débordement sur la concaténation des vecteurs de pavage et d'ajustage. Si le modulo est inutile, le code généré pour l'accès aux points sur ce tableau peut être simplifié.

Si le modulo est utile pour certains accès au tableau, il est toujours possible de générer le test d'utilité du modulo pour chacun des motifs d'accès au tableau ; on ne considère que les vecteurs d'ajustage (leur « encombrement » est calculé une fois pour toutes) et on ajoute le décalage dû à l'itération courante du pavage. On peut ensuite aiguiller la gestion du motif sur une fonction d'accès avec ou sans modulo. Notons que ce mécanisme entraîne une duplication de code, comme illustré figure 2.11.

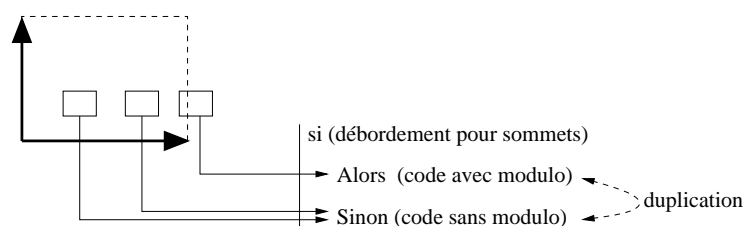


FIG. 2.11 – Utilisation conditionnelle du modulo

L'absence de modulo permet non seulement de gagner sur l'opération de division mais aussi de ne plus travailler avec des vecteurs dans le cas où le stockage des tableaux est connu comme étant linéaire. En effet, la seule opération non linéaire dans le calcul des points est l'opération de modulo, si elle n'est plus nécessaire on peut alors distribuer l'opération de linéarisation (produit scalaire) et les vecteurs de pavage et d'ajustage deviennent des scalaires. Soit la projection de linéarisation du tableau en mémoire $\mathcal{L} = \left| \begin{array}{c} 1 \\ \dim_1 \\ (\dim_1 \times \dim_2) \\ \dots \end{array} \right|$, la formule d'accès au point mémoire devient (sans le modulo) :

$$\mathcal{L} \cdot (\mathcal{P} \cdot \chi_q + \mathcal{F} \cdot \chi_d + \mathcal{O}) = (\mathcal{L} \cdot \mathcal{P}) \cdot \chi_q + (\mathcal{L} \cdot \mathcal{F}) \cdot \chi_d + (\mathcal{L} \cdot \mathcal{O})$$

²En fixant le vecteur d'itération au maximum sur les valeurs positives (resp. négatives) et à zéro sur les autres.

Donc $(\mathcal{L}.\mathcal{P})$ et $(\mathcal{L}.\mathcal{F})$ deviennent des matrices ligne (i.e. des produits scalaires) et $(\mathcal{L}.\mathcal{O})$ devient un scalaire.

2.3.4 Limitations du schéma de compilation

Le langage ARRAY-OL permet de spécifier très simplement des opérations régulières et massivement parallèles sur des tableaux. Ce genre de transformation est bien adapté au traitement de signal. De plus il se prête fort bien à une implémentation à base de nid de boucles. Cependant une telle compilation n'est pas satisfaisante en pratique.

► Unité d'une tâche

La production du code des tâches est telle qu'elle constitue une unité indivisible de calcul : une tâche doit être effectuée du début à la fin avant que la suivante ne commence. Ce modèle d'exécution nécessite le calcul complet des tableaux opérands avant leur utilisation et entraîne celui des résultats pour que la tâche finisse.

Dans une chaîne sonar réaliste, 10 itérations de la VBL sur 512 hydrophones sont nécessaires toutes les 5 secondes pour permettre une exploitation ultérieure par le traitement de données intensif. L'exécution suivant le modèle séquentiel de compilation de ce quantum nécessite 3,16 Go de mémoire pour l'allocation des tableaux. Cette taille mémoire dépasse généralement l'espace mémoire moyen d'une station de travail classique (*swap* compris) et s'approche d'ailleurs de la limite d'adressage des processus sur architecture 32 bits.

Devant l'importance des volumes de données manipulées par les applications de traitement de signal classiques, il est essentiel d'allouer ces tableaux au plus tard et de les libérer au plus tôt. Cependant, cette méthode est inopérante pour les cas dans lesquels la taille des tableaux est très hétérogène, ce qui arrive fréquemment en TS où la quantité de données diminue fortement le long de la chaîne. Ainsi l'allocation des deux premiers tableaux de la VBL (sur les dix nécessaires à l'application) occupe à elle seule 90% des ressources requises. On est alors obligé de passer par une utilisation exclusivement externe des tableaux, comme le système de fichiers par exemple, ce qui entraîne évidemment une chute dramatiquement des performances de l'exécution.

D'autres stratégies d'exécution sont donc à envisager. Ainsi une stratégie d'exécution « pipeline » consisterait à déclencher une tâche sur une partie de ses opérands pour produire une partie de ses résultats qui seront consommés par la tâche en aval. Une telle exécution ne nécessite plus une allocation complète des tableaux intermédiaires.

Pour mettre en place cette méthode, il faut, au préalable, calculer les caractéristiques du « pipeline ». On peut alors remarquer que son schéma d'exécution revient à considérer un niveau de hiérarchie ARRAY-OL dont le rôle serait de paver les tableaux par les « méta-motifs » d'exécution du « pipeline ».

► Flux infini

D'abord, on peut remarquer que les contraintes résultat nous impose l'équivalence entre une tâche infinie et un tableau résultat infini. Ces mêmes contraintes n'interdisent pas à un tableau d'entrée infini d'être utilisé par une tâche fini mais cela est fort improbable puisque cela signifierait l'utilisation d'une partie infime du tableau. Enfin il est, par contre, tout à fait envisageable pour une tâche infinie d'avoir un tableau d'entrée fini. Cela pourrait correspondre à une tâche de génération de données (par exemple, aléatoire) avec les valeurs d'entrées comme argument.

Pour en revenir à la compilation, dans le cas de tableaux infinis, l'implémentation des tâches de l'application a toute les chances de contenir une boucle sans fin. Pour être assuré de produire tous les résultats, il faut évidemment qu'il n'y ait qu'une seule tâche et que la boucle d'itérations infinie soit à l'extérieur du nid de boucles.

Maintenant nous pouvons envisager aussi que l'application n'effectue qu'un certain nombre d'itérations sur cet axe (fixé par un paramètre). Cela nous dégage des deux contraintes ci-dessus mais le problème est alors reporté sur la détermination du nombre d'itérations à effectuer sur chaque tâche et de la taille des tableaux utilisés, sachant qu'il faut que les points produits sur un tableaux intermédiaire soient suffisants pour la tâche suivante. Enfin on peut entourer cette application d'une boucle infini effectuant ces *méta-itérations* les unes après les autres. On s'aperçoit que notre application principale devient une sous application de cette nouvelle application principale qui n'a qu'une seule tâche.

On remarque alors que ce problème n'est qu'un cas particulier de la stratégie du « pipeline » que nous venons juste de mentionner. Sauf que dans ce cas, la motivation n'est plus seulement la performance mais tout simplement la faisabilité.

► Extension du compilateur

En conclusion, nous avons décidé de conserver le schéma de génération de code tel que nous l'avons présenté et de concentrer les efforts de compilation sur la modification du source ARRAY-OL dans la direction indiquée plus haut. Une autre raison pour adopter cette stratégie est qu'elle est plus ou moins indépendante du support de compilation et on a toutes les raisons d'espérer qu'elle serve aussi à une compilation sur l'architecture dédiée qu'à développée TMS. Enfin, cela permettra un meilleur retour des informations de compilation pour l'utilisateur puisqu'il pourra visualiser le source ARRAY-OL modifié.

2.4 Les environnements existants

Dans cette partie, nous présentons quelques environnements de programmation et de compilation pour le parallélisme de données dont les objectifs ne sont pas très éloignés de nos préoccupations. Nous décrirons pour chacun, le formalisme utilisé, les transformations (automatique ou non) qu'ils permettent ainsi que leur interface utilisateur.

2.4.1 PEI

PEI [VP92] est un langage déclaratif permettant de définir et de transformer des programmes data-parallèle en s'appuyant sur des équivalences sémantiques. Il a été initialement créé par E. Viollard et G.-R. Perrin en 1993 puis complété par S. Genaud.

Définition Un objet PEI est appelé *champ de données*, il est composé de deux fonctions

$$\begin{aligned}\mu &: E \subset \mathbb{Z}^n \longrightarrow V \\ \sigma &: F_n \subset \mathbb{Z}^n \longrightarrow F_p \subset \mathbb{Z}^p\end{aligned}$$

avec V un multi-ensemble de valeurs ($\wp(\mathbb{Z}), \wp(\mathbb{R})\dots$), σ bijectif et $E \subset F_n$. On a $\text{dom}(\mu) = E$, $\text{dom}(\sigma) = F_n$ et $\text{Im}(\sigma) = F_p$.

$\text{dom}(\mu)$ représente le référentiel du champs et μ permet d'indexer les données dans ce référentiel. σ peut alors être vu comme un placement des données.

Exemple : Soit le tableau $A(0 : n - 1, 0 : m - 1)$ stocké linéairement, on construit le champs de données $X_A = (\mu, \sigma)$ avec

$$\begin{aligned}\mu &: [0, n - 1] \times [0, m - 1] \longrightarrow V \\ \sigma &: [0, n - 1] \times [0, m - 1] \longrightarrow [0, nm - 1]; (i, j) \mapsto (im + j)\end{aligned}$$

Opérations Par souci de simplicité, il n'existe que 5 types d'opérateurs :

- opération fonctionnelle (\triangleright) : appliquer une opération sur toutes les données. Soit $f : \text{Im}(\mu) \subset \text{dom}(f) \longrightarrow W$, $f \triangleright (\mu, \sigma) = (f \circ \mu, \sigma)$. C'est l'opération data-parallèle par excellence, son rôle est celui de la fonction map d'autres langages.
- routage (\triangleleft) : changer l'indexation des données. Soit $g : \text{dom}(g) \subset \text{dom}(\sigma) \longrightarrow \text{dom}(\mu)$, $(\mu, \sigma) \triangleleft g = (\mu \circ g, \sigma)$. L'opérateur routage permet notamment de représenter les communications. Par exemple, si chaque élément d'un champ a besoin de la donnée en première ligne, on peut utiliser le routage $g : (i, j) \mapsto (0, j)$ ce qui correspond à une diffusion ou une réplication);

- changement de base ($::$) : changement réversible du référentiel. Soit $h : \text{dom}(\mu) \subset \text{dom}(h) \longrightarrow \mathbb{Z}^m$ une bijection, $h :: (\mu, \sigma) = (\mu \circ h^{-1}, \sigma \circ h^{-1})$. L'intérêt du changement de base est purement géométrique comme la transposition de matrice ;
- réduction géométrique ($;\triangleright$) : sémantiquement inverse du routage. Soit $g : \text{dom}(g) \subset \text{dom}(\mu) \longrightarrow \text{Im}(g) \subset \text{dom}(\sigma)$, $g;\triangleright(\mu, \sigma) = (\mu \circ g^{-1}, \sigma)$. Comme g n'est pas forcément bijective alors $\mu \circ g^{-1}(i) = \{\mu(j); g(j) = i\}$. Sinon on a $g;\triangleright X = X \triangleleft g^{-1}$;
- superposition ($/\&/$) : superposition des valeurs de deux champs de données. Soient $X = ((\mu, \sigma)$ et $Y = (\mu', \sigma')$, on doit avoir $\sigma_{\text{dom}(\sigma')} = \sigma'$, ou $\sigma'_{\text{dom}(\sigma)} = \sigma$. Alors $X/\&/Y = (\mu_1, \sigma_1)$ avec

$$\begin{aligned} \sigma_1 = \sigma & \quad \text{si} \quad \sigma_{\text{dom}(\sigma')} = \sigma' \\ \sigma_1 = \sigma' & \quad \text{si} \quad \sigma'_{\text{dom}(\sigma)} = \sigma \end{aligned}$$

et

$$\begin{aligned} \mu_1 = \mu & \quad \text{sur} \quad \text{dom } \mu \setminus \text{dom } \mu' \\ \mu_1 = \mu' & \quad \text{sur} \quad \text{dom } \mu' \setminus \text{dom } \mu \\ \mu_1 = \mu; \mu' & \quad \text{sur} \quad \text{dom } \mu \cap \text{dom } \mu' \end{aligned}$$

Environnement Du point de vue pratique, l'environnement PEI d'édition et de transformation a été conçu à partir de CENTAUR [BCD⁺89] et MAPLE. Il existe, de plus, un outil graphique, VPEI, permettant de visualiser les relations géométriques des objets d'un programme.

Le langage PEI offre un formalisme data-parallèle distinguant nettement la partie algorithmique (opération fonctionnelle, routage) de la géométrie des données. Son objectif est de permettre la modélisation et la transformation d'une large palette de langages (C^* , HPF...). Pour cela, des règles d'équivalences (faibles et fortes) entre programmes sont proposées ainsi que des stratégies de raffinement (passage de communications de voisinage à une diffusion...) [Gen97].

La direction principale de transformation consiste à reformuler les équations par les règles d'équivalence en essayant d'y introduire des opérateurs précis. On peut citer comme exemple paradigmatique le remplacement de la récursivité d'un algorithme par l'utilisation d'un opérateur de *scan*.

2.4.2 ALPHA

ALPHA [VMQ91] est un langage fonctionnel data-parallèle conçu par P. Quinton et C. Mauras [Mau89] en 1989. puis étendu par S. Rajopadhye, T. Risset depuis 1994. Un programme est décrit par un système d'équations affines récursives, SARE (figure 2.13.

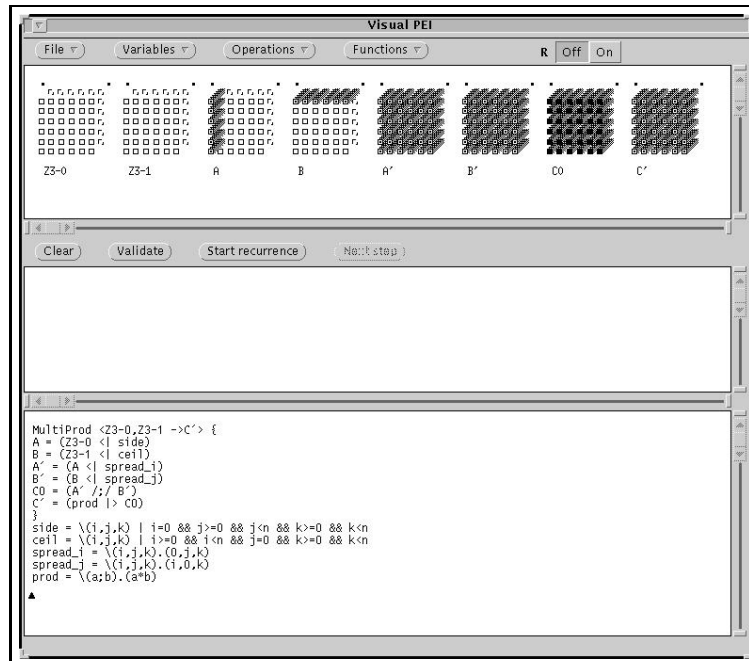


FIG. 2.12 – Exemple du produit de matrices en PEI

$$F(0 \leq i < n) = \begin{cases} 0 & \longrightarrow y \\ 1 & \longrightarrow z \\ i > 1 & \longrightarrow F(i-1) + F(i-2) \end{cases}$$

FIG. 2.13 – Exemple de la suite de Fibonacci en ALPHA

ALPHA est surtout axé sur la description et l'analyse d'algorithmes systoliques. Les valeurs non récursives représentent les entrées et les dépendances récursives les liens de communications. Enfin les dernières valeurs calculées (au sens des dépendances) sont généralement associées aux sorties.

Il existe un environnement de programmation du langage ALPHA : MMALPHA. Le moteur interne est écrit en C et il utilise MATHEMATICA pour son interface utilisateur. Plusieurs manipulations et transformations sont possibles à travers cet outil : changement de base, *pipelining* des constantes, changement de base, simplification de la forme, ordonnancement, simulation... Les transformations sont commandées à la main (fonction MATHEMATICA) mais certaines d'entre elles sont assez générales (ordonnancement, production de nids de boucles, allocation des tableaux...).

L'objectif majeur de ce projet est la synthèse d'architecture systolique (VLSI) grâce à l'extension ALPHARD (synthèse de composants réutilisables). Mais si on considère le modèle systolique comme un modèle éventuel de programmation parallèle, il est alors tout à fait possible de produire du code.

La spécification des domaines et des transformations s'exprime grâce au formalisme polyédrique (cf. 2.5.2). Cet environnement utilise la POLYLIB, écrite par H. Le Verge et D. Wilde [Wil93], comme outils de calcul. Le projet ALPHA est d'ailleurs à l'initiative du développement de cette bibliothèque.

2.4.3 Méthodologie pour le HLS

Un des axes du groupe *High Performance Computing and Network* de l'ENEA est la synthèse d'architecture de haut niveau [MNP⁺96]. En particulier A. MARONGIU et P. Palazzari travaillent sur une méthodologie de compilation de systèmes d'équations affines (SARE). Leurs travaux et leurs motivations sont très proches de ceux du projet ALPHA.

Leurs formats d'entrée sont, soit un programme directement au format SARE, soit un programme écrit en *lighthC*. Ce dernier est un sous-ensemble du langage C dont les structures de contrôle sont réduites aux nids de boucles et aux affectations. Il est donc aisé de tirer de tels programmes une forme SARE.

À partir de la description SARE, ils proposent une méthodologie [MPCM00] pour arriver jusqu'à une implémentation. Cela passe par la recherche de fonctions d'allocation et d'ordonnancement sur machine virtuelle, puis par l'application d'une opération de *clustering* pour s'adapter à l'architecture cible, et enfin la génération de code sous forme VLSI.

Ils ne donnent pas d'algorithme pour trouver les matrices d'allocation et d'ordonnancement mais proposent plutôt différentes techniques pour évaluer et optimiser une solution [MP99a, MP99b]. Ils ont ainsi développé un outil de parallélisation basé sur le modèle polyédrique et en particulier sur la POLYLIB.

2.4.4 PIPS

L'environnement PIPS [IJT91] est celui qui se rapproche le plus de notre travail. En effet, il propose des analyses et des transformations de programmes. Ceux-ci sont normalisés dans le modèle polyédrique. Les transformations doivent amener à un code compilé efficace (parallélisation, élimination des redondances...). Il a été conçu initialement par F. Irigoien, R. Triolet et P. Jouvelot en 1989.

Cet environnement a pour but l'analyse et la transformation d'applications scientifiques et de traitement de signal. Son principal avantage, comme l'indique son nom (Parallel Inter-Procedural System), est qu'il ne se restreint pas à l'étude d'une portion du code mais considère les relations inter-procédurales. Il permet donc à l'utilisateur d'étudier globalement l'impact d'un programme en mémoire et en temps, et d'exploiter ces résultats pour appliquer des transformations telles que des optimisations de code et surtout de la parallélisation.

Les formats externes sont presque exclusivement reliés au FORTRAN. L'entrée est en FORTRAN77 plus d'éventuelles directives HPF. Hormis les sorties de l'analyseur (graphe d'appel, graphe de dépendances...), le code généré est du FORTRAN77 associé à des bibliothèques de communication (PVM, MPI) ou des directives parallèles (CRAY, OpenMP).

La partie la plus importante de l'analyseur concerne évidemment l'utilisation des données, et plus particulièrement des tableaux (privatisation de variables, tableaux ou de parties de tableaux, le calcul des dépendances...). Le reste a trait aux structures de contrôle (graphe d'appel, complexité, détection des réductions...).

Parmi les opérations de modifications, on trouve les restructurations qui ne changent pas sémantiquement le code mais correspondent plutôt à une épuration de celui-ci (élimination de code mort, duplication de code...). Enfin, les transformations sont d'un plus haut niveau et modifient la structure des données et du code. Elles partent des instructions, avec la propagation des constantes et l'optimisation des expressions, jusqu'à la parallélisation du programme (on retrouve en particulier la plupart des algorithmes de parallélisation connus comme ceux d'Allen & Kennedy [AK01] et de Feautrier [Fea96]), en passant par une série d'opérations sur les nids de boucles (permutation d'indices, déroulage de boucle, *strip-mining*). On peut noter, au passage, que certaines transformations sont délicates puisqu'elles ne vérifient la validité du code modifié par exemple le *loop-nest tiling*, l'utilisateur doit s'être assuré préalablement que les paramètres qu'il a donnés sont cohérents.

Cet environnement s'appuyant fortement sur la POLYLIB, le format interne des données se base sur le modèle polyédrique. Plus précisément, le cœur des procédures d'analyse et de transformation repose sur les calculs des régions caractéristiques d'une fonction (ou d'une instruction). Les 4 types de régions sont READ/WRITE pour l'effet interne de la fonction et IN/OUT pour les dépendances avants et arrières extérieures à la fonction. Elles donnent des indications sur les durées et les fréquences de validité des

données.

Un autre des intérêts de PIPS est de proposer un atelier graphique permettant d'appliquer les transformations du code ainsi que de visualiser le résultat de certaines analyses.

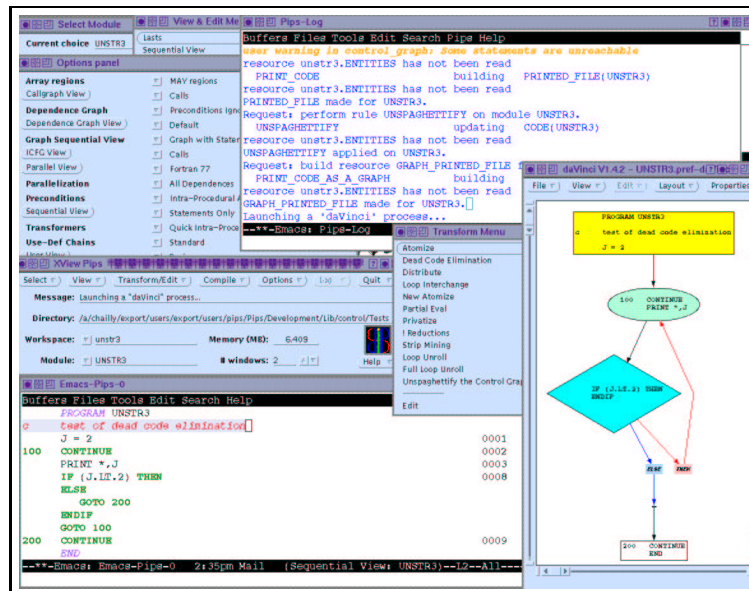


FIG. 2.14 – Vu de l'interface graphique de PIPS

2.4.5 PLC pour la compilation d'application TS

Bien que ces travaux aient été menés par C. Ancourt [ABG⁺97] en 1997 avec plusieurs personnes de l'équipe PIPS dont F. Irigoin, ils ne sont pas actuellement intégrés à cet environnement.

Ces travaux sont dédiés spécifiquement aux applications de traitement de signal numérique (DSP). Les hypothèses sur la structure des algorithmes suivent exactement celles du langage ARRAY-OL puisque ces recherches sont également issues d'une collaboration avec TMS.

L'objectif est de trouver une distribution des calculs et des données qui puissent s'exécuter en SPMD et satisfaire un ensemble de contraintes dynamiques et physiques (tailles mémoires, latence, nombre de processeurs...). La forme de la distribution des itérations d'une tâche est fixée par la formule $i = LPx + Ly + z$ où le triplet (x, y, z) représente à une permutation près le numéro de processeur, l'indice de cycle (le seul à pouvoir être infini) et l'indice d'itération. L'ordonnancement des cycles des différentes tâches est choisie de forme affine.

La présence des contraintes physiques rendent ces formules non linéaires, la résolution du système se fait alors grâce à un modèle concurrent de programmation logique par contrainte (CCLP). Ce modèle se base sur les mêmes principes que le modèle de résolution logique classique mais appliqué à des domaines différents et utilisant une autre méthode que l'unification (algèbre linéaire, arithmétique de Pressburger...). Le choix dans ce projet s'est porté sur la création d'une nouvelle classe de CCLP, Meta(F), incluant un système de résolutions polynomiale.

L'intérêt d'une telle méthode est qu'elle permet d'unifier en un seul système les contraintes et les équations linéaires (partitionnement et ordonnancement). On bénéficie d'une résolution globale du problème en alliant les méthodes les plus appropriées pour chacun des domaines considérés (système logique et calculs linéaires).

Leur processus de transformation ne modifie pas la structure des données, il s'agit seulement de trouver un ordonnancement et une distribution adéquate. Il suggère d'ailleurs que des opérations de restructuration pourraient être appliquées préalablement pour obtenir une meilleure localité et un meilleur degré de parallélisme, ce qui est plus ou moins notre stratégie. *A priori*, les deux outils pourraient donc être utilisés conjointement.

2.4.6 SUIF

Le compilateur SUIF [HAA⁺96] (Stanford University Intermediate Format) est développé depuis 1994 par le *Stanford Compiler Group* sous la direction de M. LAM.

Il ressemble fortement à PIPS dans son concept et ses motivations. Les formats d'entrée/sortie sont le langage C et le FORTRAN ; son moteur interne permet de faire des analyses de code (dépendances d'instruction dans des nids de boucles) puis d'appliquer des transformations. Celles-ci peuvent être l'échange de boucles ou la fusion de boucles ainsi que la détermination de placement et de ordonnancement parallèles.

Le partitionnement affine constitue le cœur de l'infrastructure. Il s'agit de trouver une assignation linéaire des instructions (dans le temps ou dans l'espace) en fonction de leur indice de boucle. Toute la difficulté repose sur les contraintes qui sont imposées à ce partitionnement. Il s'agit, d'une part, des contraintes sémantiques (dépendances de calculs), incontournables, et, d'autre part, des contraintes d'optimisation (degré de parallélisme, réduction des communications...). Le calcul de tel partitionnement est mené grâce à l' Ω -LIBRAIRIE (voir en section 2.5.3), un outil d'algèbre linéaire qui utilise l'arithmétique de Pressburger comme formalisme.

Il existe une nouvelle version du compilateur, SUIF 2, dont le principal intérêt réside dans la modularité des composants (analyse, génération...) et dans la possibilité d'étendre le format des *représentations intermédiaires* que manipulent ces composants. Cela permet d'utiliser différentes structures sémantiques de programmes avec n'importe quels composants.

2.4.7 Conclusion

La grande motivation de nos travaux est de rester le plus possibles dans le domaine ARRAY-OL. Les raisons en ont été expliquées au chapitre précédent. Elles tiennent autant au fait que ce langage est particulièrement adapté à l'architecture physique que TMS a développé, et que les concepteurs d'application veulent garder la main sur le processus de compilation.

La plupart de ces langages ou méthodes présentées cherchent, en premier lieu, à exhiber du parallélisme à partir de code séquentiel. Or le parallélisme dans ARRAY-OL est explicite et généralement très important, le problème vient donc de son exploitation. De plus, ils prennent souvent comme base des dépendances ponctuelles, ils considèrent les relations de l'index d'une affectation entre les différentes itérations de boucles. Tout le problème dans ARRAY-OL vient justement des cas où les tableaux sont produits par motif non unitaire (et implicitement indivisible) et consommé par des motifs non unitaires différents. On peut contourner cela en décrétant que chaque point du motif est fonction de tous les points des motifs entrants (c'est d'ailleurs ce qui est fait dans notre formalisme), on est alors obligé de trouver un moyen pour regrouper les points de sortie ayant les mêmes points d'entrée sous peine d'avoir une explosion des redondances. C'est pourquoi il ne nous semble pas inutile de faire un système *ad-hoc*.

2.5 Modèles linéaires et outils

Les environnements que nous avons cités ainsi que nos travaux manipulent et transforment quasi-exclusivement des structures de données linéaires (tableaux, vecteurs...) et pour se faire utilisent des formalismes adéquates. Nous allons maintenant présenter quelques-uns de ces modèles.

2.5.1 Caractéristiques du domaine d'étude

On se focalise sur le traitement numérique intensif. Dans ce cas les formats d'entrée (FORTRAN, ARRAY-OL) décrivent le plus souvent les traitements par des itérations de boucles imbriquées (nids de boucles) et plus particulièrement, les itérations de boucles servent à parcourir les éléments des tableaux à calculer. Les critères spécifiques sont donc les bornes d'itérations des boucles et les fonctions d'indexage des éléments.

La première caractéristique de ces fonctions est qu'elles travaillent sur des entiers. Ensuite, sur la forme des fonctions, l'écrasante majorité des modèles se basent sur l'algèbre linéaire. Au delà du fait que ce cas est assez répandu en pratique, cette hypothèse s'explique par l'abondance de résultats théoriques dans ce domaine.

Sorti de ce contexte, on peut encore trouver quelques travaux formels sur les fonctions quadratiques [Li01] mais l'essentiel consiste en heuristiques de recherches de solu-

tions approchées. Puisque, comme nous le verrons, ARRAY-OL appartient au domaine linéaire, nous ne nous étendrons pas sur les autres modèles.

Enfin l'intérêt de pouvoir travailler avec des paramètres supplémentaires (les fonctions restant linéaires sur ceux-ci) permet de trouver des solutions valables pour du code paramétré. Il permet aussi d'unifier les questions statiques en considérant, par exemple, qu'un sous-nid est paramétré par les indices des boucles supérieures.

2.5.2 Les modèles formels

Dans les modèles présentés ci-dessous, les espaces de travail sont des espaces affines dont les dimensions sont la réunion des paramètres et des variables d'itérations. Le but est alors de décrire l'influence d'un groupe d'instructions en analysant les indices des éléments mis en jeu au moyen de parties d'espaces ou de relations entre espaces.

► Les polyèdres

L'objet formel couramment employé pour représenter les objets décrits au dessus est le polyèdre. Il s'applique généralement sur des corps tels que \mathbb{Q} ou \mathbb{R} . On peut le décrire comme l'ensemble des points satisfaisant un système d'inégalités affines. Du point de vue géométrique, il s'agit de l'intersection de semi-espaces délimités par des hyperplans (un hyperplan représente l'égalité d'une équation linéaire et les deux semi-espaces correspondent aux deux sens de l'inégalité). Il existe une forme duale (grâce à l'utilisation du lemme de FARKAS [Dax97]) plus constructive dans laquelle le polyèdre est décrit par un ensemble de sommets (points), de rayons (droites unidirectionnelles) et de lignes (droites bidirectionnelles) : l'enveloppe convexe de l'ensemble des sommets forme un polyèdre fini (polytope), les rayons l'étendent à l'infini dans une direction (cône) et enfin les lignes dans les deux directions.

Ainsi on peut représenter l'ensemble des itérations d'un nid de boucles comme les points entiers d'un polyèdre et l'image de ce polyèdre par la fonction affine d'adressage donne l'enveloppe des éléments pris en compte. Elle peut en contenir plus que nécessaire si la fonction appliquée n'est pas inversible entière (condition pour conserver un polyèdre entier) comme illustré sur la figure 2.15.

► Les matrices linéairement bornées (*Linearly Bounded Lattice*)

Le problème des polyèdres est qu'ils sont convexes et qu'ils travaillent sur un corps dense convexe alors que les ensembles entiers considérés sont souvent creux. Comme expliqué ci-dessus, l'image linéaire d'un polyèdre entier n'est pas en général l'image entière du polyèdre. Pour pallier cet inconvénient, on ajoute des dimensions dont le but est de rendre compte de ces creux. Par exemple, l'ensemble des entiers pairs entre 0 et

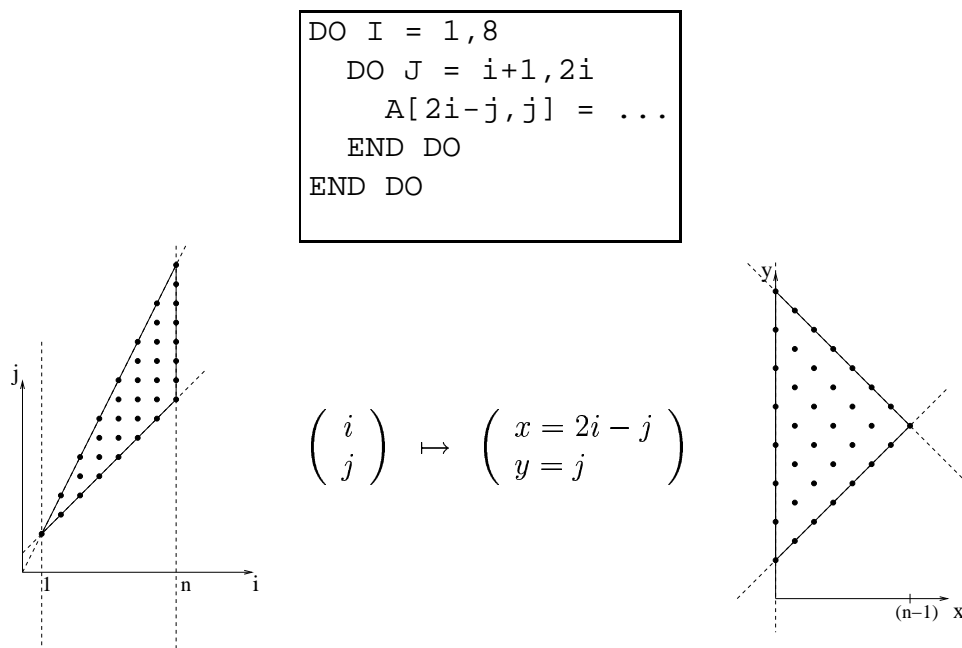


FIG. 2.15 – Le plan de gauche représente l'ensemble des itérations (i, j) de la boucle ci-dessus. Il s'agit des points entiers contenu dans le polyèdre dessiné. La formule médiane donne la fonction de passage des indices d'itérations de boucle aux indices du tableau A dans l'itération. Le plan de droite représente l'ensemble des indices du tableau A affectés par la boucle entière. On remarque que cet ensemble est bien inclus dans l'image du polyèdre mais qu'il est creux i.e. on ne retrouve qu'un point entier sur deux.

$2n$, peut être décrit par un polyèdre entier qu'on projette sur l'axe i (cf. figure 2.16). Ceci correspond exactement à la définition d'une LBL ($\{x \in \mathbb{Z}^n; \exists y \in \mathbb{Z}^m, Ax + By \leq c\}$) i.e. la projection des points entiers d'un polyèdre sur un sous-espace.

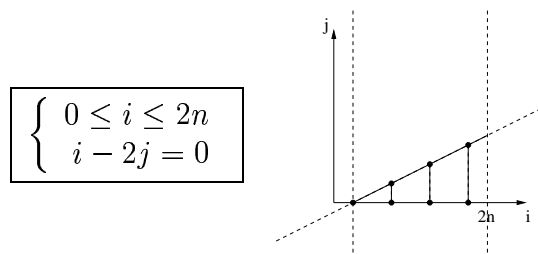


FIG. 2.16 – Le polyèdre du système à 2 équations correspond à la droite de pente $1/2$ sur le plan. Sa projection entière sur i ne laisse que les entiers pairs.

Ce modèle se rapproche plus des structures réellement concernées mais l'utilisation d'un ensemble qui n'est pas un corps (\mathbb{Z}) oblige à manipuler des expressions non linéaires comme des divisions entières ou des modulus.

► L'arithmétique de Pressburger

L'arithmétique de Pressburger [GS66] se base sur la combinaison de contraintes linéaires avec la logique du premier ordre i.e. des connecteurs logiques (\wedge, \vee, \dots) et les prédicats existentiels et universels (\exists, \forall).

Elle est équivalente à celle basée sur les polyèdres. Ces deux modèles représentent plus ou moins la dualité entre une forme ensembliste et une forme propositionnelle (\wedge pour l'intersection, \vee pour l'union...).

2.5.3 Les outils existants

On peut classer les différents outils existants selon, d'une part le modèle qu'ils utilisent et, d'autre part, les opérations qu'ils proposent.

Pour les polyèdres, la référence est sans aucun doute la POLYLIB développé par D. Wilde. Les polyèdres possèdent des caractéristiques mathématiques permettant de les manipuler géométriquement, on peut en faire l'intersection, en trouver l'image par une fonction affine... Cette bibliothèque permet d'appliquer ces opérations sur des unions de polyèdres (qui ne sont pas des polyèdres) paramétrés. Elle propose également, grâce au polynôme de Ehrhart une opération de plus haut niveau : compter le nombre de points entiers dans une union de polyèdres. La POLYLIB est largement utilisée dans plusieurs environnements (ALPHA, PIPS...).

Le programme PIP [Fea88] écrit par Paul Feautrier ne propose qu'une seule opération : la résolution du simplex i.e. trouver le point d'un polyèdre qui minimise une

fonction linéaire de coût. Toute la difficulté réside dans le fait que cette résolution opère dans l'espace des entiers (le polyèdre est, en fait, une LBL) et avec des paramètres. Il répond alors par un QUA_{ST} (*QUasi-Affine System Tree*) qui est une structure arborescente dont les nœuds sont des inéquations sur les paramètres et les feuilles donnent l'expression du minimum en fonction de ceux-ci. Ce minimum est valable pour les valeurs des paramètres contenues dans le polyèdre découlant des inéquations de la racine à la feuille.

Ce programme a été initialement écrit pour résoudre les problèmes de dépendances d'instructions dans les boucles : en effet, la valeur d'un élément de tableaux dépend de la dernière instruction l'ayant modifié. En exprimant par un polyèdre, l'ensemble des valeurs d'itérations dans lesquelles cet élément est modifié, PIP permet d'en extraire la plus grande (la dernière par ordre chronologique) et donc celle qui est responsable de la valeur finale de l'élément.

Cependant la fonction de PIP n'étant ni plus ni moins que la résolution du simplexe paramétré en points entiers, il est donc normal que son application se soit étendue à diverses domaines de la parallélisation automatique comme l'ordonnancement [BDSV97], la génération de code [Bou96], ou l'évaluation de communication HPF.

Enfin il existe également une autre bibliothèque, l' Ω -LIBRAIRIE [KMP⁺96], plus ou moins équivalente à la POLYLIB. Elle admet à peu près les mêmes manipulations géométriques et propose quelques transformations supplémentaires comme la fermeture transitive d'une relation et surtout la simplification des expressions (Ω -test [Pug92]). La différence majeure est qu'elle se base sur l'arithmétique de Pressburger. Cette bibliothèque est le support de la plate-forme Ω [KP93a] dont le rôle est de permettre l'analyse et la transformation de programmes.

2.5.4 Conclusion

Le programme PIP était bien adapté au problème pour lequel il avait été conçu, il a depuis été utilisé dans pour bien d'autres applications. Or les équations de ces dernières contiennent souvent des divisions euclidiennes, des modulus et des parties entières. Ces opérateurs sont remplacés en internes par de nouvelles variables et de nouvelles équations faisant apparaître explicitement les divisions ($i \equiv x(n) \Leftrightarrow i = x + kn$). Il se trouve hélas que leur présence a tendance à produire des réponses très complexes et difficilement exploitables.

À noter l'existence du projet SPOCs [BR99] qui propose une interface unifiée et agréable aux trois outils précédemment cités et surtout s'efforce de simplifier les formules en vue de pouvoir enchaîner les calculs sans que l'expression des résultats explose.

Il existe de nombreuses similitudes entre les spécifications d'ARRAY-OL et les modèles déjà présentés, en particulier les polyèdres (linéarité des relations, la notion d'en-

semble d'itérations...). D'un côté, le modèle ARRAY-OL est plus simple dans le sens où les nids de boucles sont parfaits, non paramétrés et totalement indépendants. De l'autre, l'omniprésence des modulo implique des divisions entières qui cassent une partie de la linéarité et qui sont assez mal gérées par les environnements actuels.

On se retrouve alors devant le choix d'utiliser :

- soit un formalisme très puissant et pour lequel il existe plusieurs outils. Dans ce cas, les réponses obtenues utiliseraient toutes les possibilités d'expression des polyèdres, il faudrait contraindre ces résultats à prendre la forme du modèle ARRAY-OL ;
- soit un formalisme *ad hoc* qui ne bénéficie pas de tous les études théoriques mais qui nous permet d'assurer que le résultat des transformations ne sera jamais très éloigné de la forme recherchée.

Nous avons choisi la deuxième approche en utilisant le modèle des ODT décrit au chapitre 3 ci-après. Il apparaîtra de manière évidente que, d'une part le formalisme qui nous utiliserons est assez proche et équivalent à ceux que nous avons présentés ici, et que d'autre part, les résultats mathématiques qui nous seront utiles sont ceux sur lesquels reposent les formalismes présentés.

Chapitre 3

Modification du schéma d'exécution par hiérarchisation

Suite à l'expertise du compilateur actuel, nous avons proposé des modifications ponctuelles du code généré. Afin d'obtenir un compilateur supportant effectivement l'exécution de tout programme ARRAY-OL, des transformations plus fondamentales sont nécessaires. Nous proposons une modification du schéma de compilation du code ARRAY-OL par transformations au niveau du code source ARRAY-OL mais sans toucher au schéma d'exécution déjà décrit. Un formalisme de distribution de tableaux (ODT) a été défini, il aboutit à une spécification exploitable de ces transformations afin de les automatiser.

3.1 Nouvelle approche d'exécution

3.1.1 Schéma d'exécution direct

Le compilateur séquentiel évoqué au chapitre précédent implémente un schéma d'exécution très simple puisqu'il considère qu'une tâche constitue une unité indivisible du calcul ce qui implique la complétude du tableau opérande avant le début de celle-ci et la complétude du résultat en fin de tâche. Cette mauvaise gestion mémoire rend le code généré limité, tant du point de vue de la faisabilité (taille des applications pouvant être exécutées) que de l'efficacité (performances de l'exécution des applications ARRAY-OL). Pourtant elle reste très facile à mettre en œuvre et efficace du point de vue algorithmique.

Trois types d'utilisation des tableaux coexistent : les tableaux opérandes de l'application, les tableaux résultats de l'application, et les tableaux intermédiaires utilisés entre deux tâches de l'application. Hormis lors du débogage, seuls les tableaux d'entrée et de sortie intéressent l'utilisateur. Les tableaux intermédiaires ne servent qu'à stocker

des calculs temporaires. Dans l'état actuel du code généré, l'espace mémoire alloué à un tel tableau n'est réutilisé qu'après la terminaison de la tâche, c'est-à-dire après le remplissage complet du tableau.

Une solution serait de calculer ces tableaux intermédiaires morceau par morceau, de consommer le plus rapidement possible chaque morceau dans la tâche suivante (transversalité de l'exécution) et ainsi pouvoir utiliser l'espace mémoire associé à un autre morceau.

3.1.2 Schéma d'exécution dynamique et parallélisme de l'exécution

Le grain d'exécution minimal d'une application ARRAY-OL est : fonctionnellement, la TE ; spatialement, le motif. Une exécution transversale maximale consisterait à calculer une TE dès que ses motifs opérands sont complets et à libérer au plus vite la place mémoire associée à ces motifs. Le bénéfice produit par cette stratégie est maximal si l'ordre de production des motifs résultats intermédiaires est tel qu'ils produisent au plus vite des motifs opérands pour les tâches suivantes.

La mise en place de cette stratégie demande de contourner les difficultés principales suivantes :

- l'identification d'un « bon » ordre de déclenchement des tâches ;
- la gestion mémoire des motifs incomplets ;
- l'identification des motifs complets permettant le déclenchement des tâches suivantes.

Une implantation dynamique de cette stratégie doit garder la maîtrise du coût du contrôle de l'application.

Par ailleurs, nous constatons qu'une tâche ARRAY-OL est intrinsèquement parallèle : chaque TE peut être exécutée indépendamment des autres. Une méthode de parallélisation en mémoire partagée venant naturellement à l'esprit consisterait à donner une partie des itérations de pavage aux différentes unités de calculs. Ce schéma nécessiterait cependant de nombreuses synchronisations entre les différentes exécutions parallèles.

Dans le cas de l'utilisation d'un réseau de stations (mémoire distribuée), les difficultés déjà évoquées sont en plus reportées au niveau du découpage des tableaux entre les différentes mémoires en accord avec les ensembles d'itérations associés aux processeurs.

Nous estimons que cette approche totalement dynamique ne permettrait pas de *rentabiliser* l'effort fourni pour la mettre en place. Nous penchons donc plutôt pour une stratégie plus *statique*.

3.1.3 Hiérarchisation

Une approche intermédiaire des deux précédentes (celle du schéma d'implémentation directe et celle décrite juste au-dessus) est de considérer un ensemble de mo-

tifs opérands permettant d'enchaîner plusieurs tâches en produisant à chaque fois un nombre entier de motifs. Les tableaux opérands sont alors découpés en *macro-motifs* : les macro-motifs sont les sous-ensembles des éléments des tableaux opérands d'une tâche permettant à cette tâche de produire au moins un motif ; chaque macro-motif se pavant et s'ajustant sur leur tableau associé. De nouvelles boucles d'itérations sur les macro-motifs assurent le traitement complet des tableaux opérands (cf. figure 3.1).

L'expression de ces boucles d'itération sur les macro-motifs repose sur l'utilisation de la hiérarchie d'ARRAY-OL.

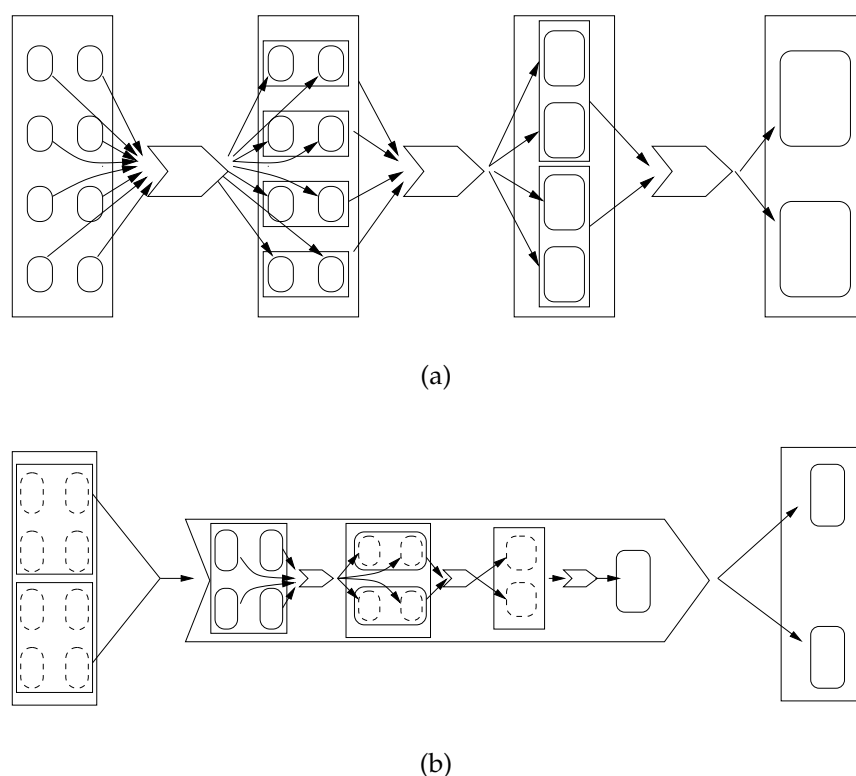


FIG. 3.1 – Réécriture (a) d'une application ARRAY-OL composée de trois tâches successives en (b) une application ARRAY-OL hiérarchique d'une tâche elle-même composée de trois sous-tâches. On note la possible réduction de la taille des tableaux intermédiaires

Cette stratégie a déjà été mise en œuvre manuellement chez TMS lors de la réécriture de codes ARRAY-OL en vue de leur implantation sur l'ACC-SYNC. Ces travaux sont la raison majeure de l'introduction de la hiérarchie dans le modèle ARRAY-OL.

Nous proposons d'automatiser ce travail de transformation de code ARRAY-OL en un code ARRAY-OL hiérarchique.

Pour ce faire, il nous faut pouvoir spécifier de manière formelle les dépendances

de calculs entre les points des tableaux mis en jeu. Celles-ci rassemblent des caractéristiques de linéarité et de contraintes. Il existe déjà, dans ce domaine, plusieurs modèles de description, cependant les fortes contraintes du langage ARRAY-OL nous ont incité à utiliser une nouvelle formalisation *ad hoc* : les ODT.

Nous avons déjà discuté des modèles déjà existants au chapitre précédent (section 2.5.2) et de l'intérêt d'utiliser un modèle *ad hoc*. Nous allons maintenant le présenter. Il nous permet de formaliser la notion de tâche ARRAY-OL au moyen d'opérateurs de relations de points entre tableaux opérandes et résultats. Nous exprimerons ensuite les transformations de code ARRAY-OL au niveau de ce formalisme.

3.2 Un formalisme approprié : les ODT

Le formalisme des ODT [Dem98] (Opérateurs de Distribution de Tableaux) a été proposé par A. DEMEURE pour spécifier les dépendances entre opérandes et résultats dans les applications ARRAY-OL. Ce formalisme pourrait amener à terme à la spécification complète d'une application ARRAY-OL et à l'expression de son placement sur une machine parallèle donnée.

Le formalisme des ODT s'appuie sur l'algèbre linéaire avec contraintes (c'est-à-dire sur des espaces bornés) et permet de définir, au travers d'un certain nombre d'opérateurs, les liens existants entre les points de deux espaces du type \mathbb{K}^n . Ces opérateurs servant à décrire les relations de dépendance de points entiers, il semble assez logique que l'espace manipulé soit \mathbb{Z} . Il en sera ainsi dans la grande majorité des cas et particulièrement pour les formes finales. Cependant, à certains moments, nous aurons besoin de nous plonger dans l'espace \mathbb{Q} pour revenir ensuite à des points entiers.

Ces opérateurs jouent le rôle de filtres : ils coupent, laissent passer ou dupliquent les liens entrants. Il ne s'agit pas de fonctions mais de relations (au sens mathématique) : un point d'entrée peut avoir zéro, un, ou plusieurs (voire une infinité de) points en sortie.

3.2.1 Notations

Dans la suite du document, nous adopterons les conventions suivantes :

- les scalaires seront en minuscules ;
- les variables désignant des vecteurs seront soit écrites en majuscule, soient surmontées d'une flèche ($M, \vec{x} \dots$) ;
- les matrices seront en majuscules calligraphiques ($\mathcal{M}, \mathcal{P} \dots$) ;
- les symboles \sim et ∞ désigneront indifféremment l'infini ;
- la matrice nulle sera notée ($\mathbf{0}$) et la matrice identité, ($\mathbf{1}$) ;
- plusieurs opérateurs ODT sont décrits par un vecteur. Pour les distinguer entre eux, ces vecteurs seront indexés par un signe ou une lettre en majuscule gras ($\mathbf{G}, \mathbf{M}, \mathbf{S}, \star, \star\star$).

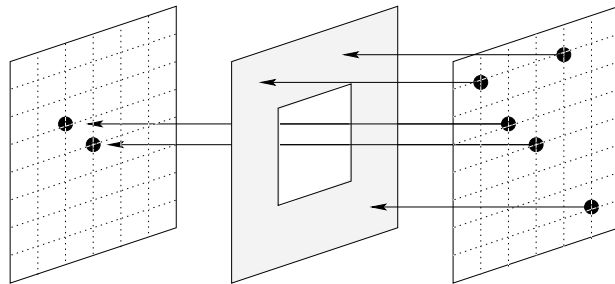
3.2.2 Les opérateurs élémentaires

Les ODT sont définis à partir de quelques relations qui forment l'ensemble des opérateurs élémentaires (OE) :

Le gabarit $\begin{pmatrix} \overrightarrow{\min}, \overrightarrow{\max} \\ \mathbf{G} \end{pmatrix}$ avec $\overrightarrow{\min}, \overrightarrow{\max} \in \mathbb{K}^n$, ou $\begin{pmatrix} \overrightarrow{\max} \\ \mathbf{G} \end{pmatrix}$ si $\overrightarrow{\min}$ est nul

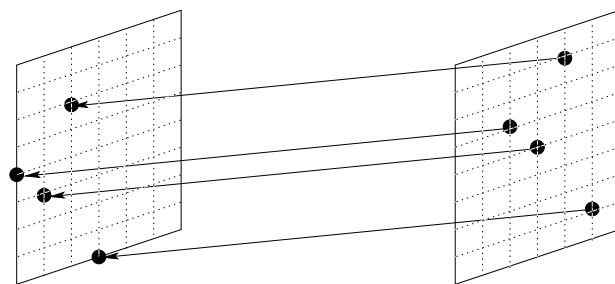
$$\begin{array}{ccc} \mathbb{K}^n & \longrightarrow & \mathbb{K}^n \\ \overrightarrow{\min} \leq \overrightarrow{x} < \overrightarrow{\max} & \mapsto & \overrightarrow{x} \end{array}$$

On remarque que certains points de départ (ceux hors de gabarit) n'ont aucun lien en sortie. De plus, un indice infini indique l'absence de contraintes sur la dimension.



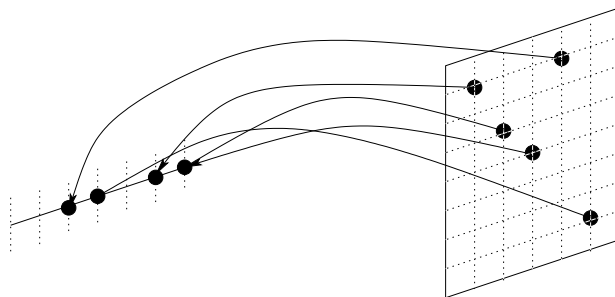
Le décalage $\begin{pmatrix} \overrightarrow{\text{shift}} \\ \mathbf{S} \end{pmatrix}$, avec $\overrightarrow{\text{shift}} \in \mathbb{K}^n$

$$\begin{array}{ccc} \mathbb{K}^n & \longrightarrow & \mathbb{K}^n \\ \overrightarrow{x} & \mapsto & \overrightarrow{x} + \overrightarrow{\text{shift}} \end{array}$$



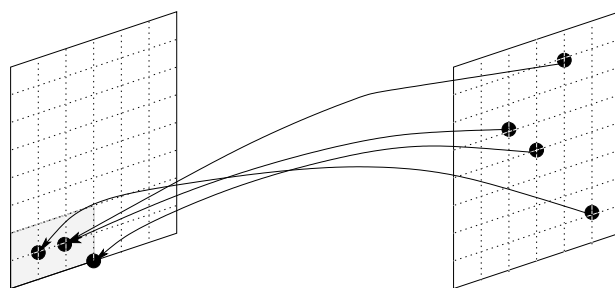
La projection $|\mathcal{M}|$ avec \mathcal{M} , une matrice $m \times n$

$$\begin{array}{ccc} \mathbb{K}^n & \longrightarrow & \mathbb{K}^m \\ \overrightarrow{x} & \mapsto & \mathcal{M} \cdot \overrightarrow{x} \end{array}$$



Le modulo $\begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix}$ avec $\vec{m} \in \overline{\mathbb{Z}}^n$

$$\begin{aligned} \mathbb{K}^n &\longrightarrow \mathbb{K}^n \\ \vec{x} &\mapsto (x_i \pmod{m_i}) \end{aligned}$$

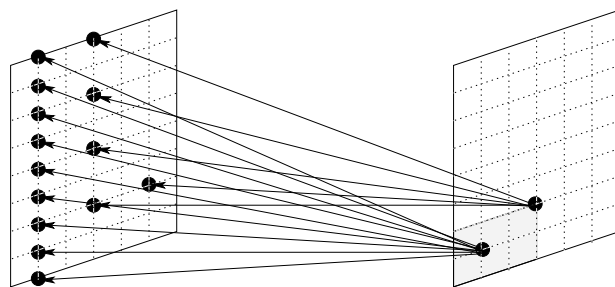


Un indice infini indique qu'il n'y a pas de modulo sur cette dimension mais qu'on ne conserve que les valeurs positives (les valeurs d'entrées négatives n'ont pas de liens).

L'éclatement $\begin{pmatrix} \vec{m} \\ \star \end{pmatrix}$ avec $\vec{m} \in \overline{\mathbb{Z}}^n$

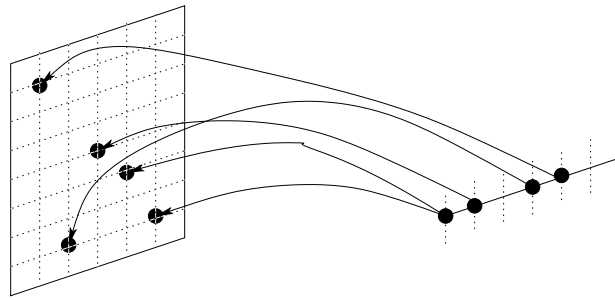
$$\begin{aligned} \mathbb{K}^n &\longrightarrow \mathbb{K}^n \\ \vec{0} \leq \vec{x} < \vec{m} &\mapsto (x_i + k_i \cdot m_i) \quad \forall k_i \in \mathbb{K} \end{aligned}$$

Un indice infini a la même signification que pour le modulo. De plus, on remarque qu'un point est lié à une infinité de points destinations.



La segmentation $\underline{\mathcal{M}}$ avec \mathcal{M} , une matrice $m \times n$

$$\begin{aligned} \mathbb{K}^m &\longrightarrow \mathbb{K}^n \\ \vec{x} &\mapsto \vec{y} \quad \text{tel que} \quad \vec{x} = \mathcal{M} \vec{y} \end{aligned}$$



L'arrondi $\lfloor \cdot \rfloor$

$$\begin{aligned} \mathbb{Q}^n &\longrightarrow \mathbb{Z}^n \\ \vec{x} &\mapsto \lfloor \vec{y} \rfloor \end{aligned}$$

On désigne par $\lfloor \cdot \rfloor$ la partie entière.

Le fractionneur $\lceil \cdot \rceil$

$$\begin{aligned} \mathbb{Z}^n &\longrightarrow \mathbb{Q}^n \\ \vec{x} &\mapsto \{ \vec{x} + \vec{r}; \vec{0} \leq \vec{r} < \vec{1} \} \end{aligned}$$

3.2.3 L'ensemble des ODT

La loi de composition naturelle sur les relations correspond à la transitivité des liens : deux points (x, y) sont reliés au travers d'une composition de relations s'il existe une chaîne de points allant de x à y telle que chaque point est lié à son suivant par une relation de la composition. On ne peut composer deux relations que si l'espace d'arrivée de la première est égal à l'espace de départ de la seconde. Nous notons cette loi de composition par un point $(.)$ et nous l'utilisons de la même façon que la loi de composition des fonctions (de droite à gauche).

Les opérateurs élémentaires et la loi de composition engendrent un ensemble de relations qui sont appelées ODT. Une composition de compositions d'OE étant une composition d'OE, la loi de composition est une loi interne sur les ODT.

► Éclatement libre

Pour simplifier les notations, on appellera *éclatement libre*, l'ODT constitué d'un modulo suivi d'un éclatement sur le même vecteur (cf. figure 3.2) :

$$\begin{pmatrix} \vec{m} \\ ** \end{pmatrix} = \begin{pmatrix} \vec{m} \\ * \end{pmatrix} \cdot \begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix}$$

Le modulo a pour effet d'annuler le rôle du gabarit implicite de l'éclatement : tous les points ayant le même résultat par modulo sont reliés.

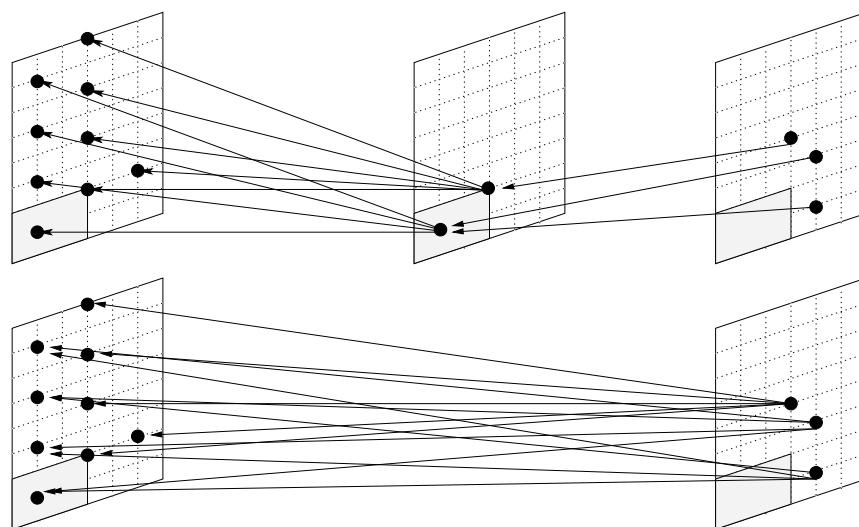


FIG. 3.2 – Exemple de composition d'OE. Un modulo suivi d'un éclatement forme un « éclatement libre »

3.2.4 ODT miroirs

Dans la suite, nous avons besoin de symétriser les ODT. Par symétrie, nous désignons le fait de construire un ODT ayant exactement les mêmes liens entre les points des deux espaces extrêmes mais en échangeant les espaces source et destination. Nous appellerons indifféremment *miroir* ou *symétrique*, l'ODT résultat de la symétrie et le désignerons à l'aide de l'exposant $(^{-1})$.

Le miroir d'une composition d'ODT s'effectue de la même manière que l'inverse d'une composition de fonction : en composant les miroirs des ODT dans l'ordre opposé

$$(\text{ODT}_1 \cdot \text{ODT}_2)^{-1} = \text{ODT}_2^{-1} \cdot \text{ODT}_1^{-1}$$

Il nous suffit donc de décrire les miroirs des OE :

Le gabarit $\begin{pmatrix} \overrightarrow{\text{min}}, \overrightarrow{\text{max}} \\ \mathbf{G} \end{pmatrix}^{-1} = \begin{pmatrix} \overrightarrow{\text{min}}, \overrightarrow{\text{max}} \\ \mathbf{G} \end{pmatrix}$

Le décalage $\begin{pmatrix} \overrightarrow{\text{shift}} \\ \mathbf{S} \end{pmatrix}^{-1} = \begin{pmatrix} -\overrightarrow{\text{shift}} \\ \mathbf{S} \end{pmatrix}$

Le modulo $\begin{pmatrix} \overrightarrow{m} \\ \mathbf{M} \end{pmatrix}^{-1} = \begin{pmatrix} \overrightarrow{m} \\ \star \end{pmatrix}$

L'éclatement $\begin{pmatrix} \overrightarrow{m} \\ \star \end{pmatrix}^{-1} = \begin{pmatrix} \overrightarrow{m} \\ \mathbf{M} \end{pmatrix}$

La projection $|\mathcal{M}|^{-1} = \overline{\mathcal{M}}$

La segmentation $\overline{\mathcal{M}}^{-1} = | \mathcal{M} |$

L'arrondi $\lfloor^{-1} = \rfloor$

Le fractionneur $\rfloor^{-1} = \lfloor$

$$\begin{aligned} \text{L'éclatement libre } \begin{pmatrix} \vec{m} \\ \star\star \end{pmatrix}^{-1} &= \left(\begin{pmatrix} \vec{m} \\ \star \end{pmatrix} \cdot \begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix} \right)^{-1} \\ &= \begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix}^{-1} \cdot \begin{pmatrix} \vec{m} \\ \star \end{pmatrix}^{-1} \\ &= \begin{pmatrix} \vec{m} \\ \star \end{pmatrix} \cdot \begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix} = \begin{pmatrix} \vec{m} \\ \star\star \end{pmatrix} \end{aligned}$$

Attention, la symétrie n'est pas, à proprement parlé, l'inversion par rapport à la loi de composition i.e. la composée d'un ODT et de son « miroir » n'est pas l'identité. On peut le vérifier, de nouveau, sur l'éclatement libre qui est la composition d'un modulo et de son miroir (un éclatement sur le même vecteur) mais qui ne donne pas l'identité (cf.figure 3.2).

3.2.5 Application d'un ODT

On rajoute, sur les ODT, une loi externe notée \blacktriangleleft . Elle permet d'appliquer un point ou un ensemble de points de l'espace d'entrée sur un ODT pour obtenir les points de l'espace de destination qui leur sont liés.

Par exemple :

$$\begin{aligned} &\begin{pmatrix} 512 \\ \sim \\ \mathbf{M} \end{pmatrix} \cdot \begin{vmatrix} 1 & 0 & 0 \\ 0 & 512 & 1 \end{vmatrix} \cdot \begin{pmatrix} 512 \\ \sim \\ \mathbf{G} \end{pmatrix} \blacktriangleleft \left\{ \begin{pmatrix} 256 \\ 8 \\ 16 \end{pmatrix} \right\} \\ &= \begin{pmatrix} 512 \\ \sim \\ \mathbf{M} \end{pmatrix} \cdot \begin{vmatrix} 1 & 0 & 0 \\ 0 & 512 & 1 \end{vmatrix} \blacktriangleleft \left\{ \begin{pmatrix} 256 \\ 8 \\ 16 \end{pmatrix} \right\} \\ &= \begin{pmatrix} 512 \\ \sim \\ \mathbf{M} \end{pmatrix} \blacktriangleleft \left\{ \begin{pmatrix} 256 \\ 4112 \end{pmatrix} \right\} \\ &= \left\{ \begin{pmatrix} 256 \\ 4112 \end{pmatrix} \right\} \end{aligned}$$

Exemple miroir du précédent :

$$\begin{aligned}
& \begin{pmatrix} 512 \\ \sim \\ 512 \\ \mathbf{G} \end{pmatrix} \cdot \frac{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 512 & 1 \end{pmatrix}}{\cdot} \begin{pmatrix} 512 \\ \sim \\ \star \end{pmatrix} \blacktriangleleft \left\{ \begin{pmatrix} 256 \\ 4112 \end{pmatrix} \right\} \\
&= \begin{pmatrix} 512 \\ \sim \\ 512 \\ \mathbf{G} \end{pmatrix} \cdot \frac{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 512 & 1 \end{pmatrix}}{\cdot} \blacktriangleleft \left\{ \begin{pmatrix} 256 + 512k \\ 4112 \end{pmatrix} ; \forall k \in \mathbb{Z} \right\} \\
&= \begin{pmatrix} 512 \\ \sim \\ 512 \\ \mathbf{G} \end{pmatrix} \blacktriangleleft \left\{ \begin{pmatrix} 256 + 512k \\ 8 + k \\ 16 - 512k \end{pmatrix} \right\} \\
&= \left\{ \begin{pmatrix} 256 \\ 8 \\ 16 \end{pmatrix} \right\}
\end{aligned}$$

3.3 ODT et ARRAY-OL

Nous allons présenter, dans cette section, comment les ODT sont reliés au langage ARRAY-OL. Nous donnerons aussi quelques propriétés sur les ODT, générales ou plus en rapport avec le langage. Nous utiliserons ces propriétés pour mener à bien les transformations.

3.3.1 Représentation d'une tâche ARRAY-OL par les ODT

La représentation d'une tâche ARRAY-OL par des ODT exhibe les liens qui existent entre les points de l'espace d'itération, nommé espace QD et ceux des tableaux.

Plus précisément, il s'agit de deux expressions construites à partir des opérateurs définissant les liens de l'espace QD vers les tableaux opérands et résultats : la première expression exprime les liens entre les tableaux opérands et l'espace d'itération ; la seconde expression exprime les liens entre les tableaux résultats et l'espace d'itération.

De manière générale, ces expressions sont constituées des opérateurs suivants : un gabarit donne les bornes d'itérations de pavage et d'ajustage ; il est suivi d'une projection constituée des vecteurs de pavages et d'ajustage et d'un modulo sur les dimensions du tableau. Il peut apparaître des opérateurs de décalage après le gabarit et avant le modulo, suivant la manière dont sont choisis les origines des tableaux.

La tâche de transformation de Fourier (FFT) de la VBL, déjà présentée en 2.2.2, se

traduit par :

$$\left\{ \begin{array}{l} \left(\begin{array}{c} 512 \\ \sim \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 512 & 1 \end{array} \right| \cdot \left(\begin{array}{c} 512 \\ \sim \\ 512 \} \text{champs } \mathbf{qd} \\ \mathbf{G} \end{array} \right) \\ \left(\begin{array}{c} 512 \\ 256 \\ \sim \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array} \right| \cdot \left(\begin{array}{c} 512 \\ \sim \\ 256 \} \text{champs } \mathbf{qd} \\ \mathbf{G} \end{array} \right) \end{array} \right.$$

Cette représentation est l'exacte traduction de l'ARRAY-OL littéral. Cependant on perd ici le lien entre les points opérands et résultats qu'est l'espace *QD*. Pour le retrouver, on ajoute les parties manquantes dans chaque gabarit (les itérations d'ajustage résultat pour l'opérande et vice versa) en les associant à des vecteurs nuls dans les projections (représentés en gras).

$$\left\{ \begin{array}{l} \left(\begin{array}{c} 512 \\ \sim \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{cccc} 1 & 0 & 0 & \mathbf{0} \\ 0 & 512 & 1 & \mathbf{0} \end{array} \right| \cdot \left(\begin{array}{c} 512 \\ \sim \\ 512 \\ \mathbf{256} \\ \mathbf{G} \end{array} \right) \\ \left(\begin{array}{c} 512 \\ 256 \\ \sim \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{cccc} 1 & 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & 1 \\ 0 & 1 & \mathbf{0} & 0 \end{array} \right| \cdot \left(\begin{array}{c} 512 \\ \sim \\ \mathbf{512} \\ 256 \\ \mathbf{G} \end{array} \right) \end{array} \right.$$

Puisque ces deux ODT ont maintenant leur espace de départ identique (l'espace *QD*), on peut symétriser l'ODT résultat et le composer avec l'opérande. On obtient alors une seule expression exprimant les liens des points résultats vers les points opérands (en toute rigueur, on devrait retrouver les gabarits identiques des deux ODT côte à côte au milieu de l'expression. Ils sont bien évidemment redondants et on n'en a gardé qu'un seul.) :

$$\left(\begin{array}{c} 512 \\ \sim \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 512 & 1 & 0 \end{array} \right| \cdot \underbrace{\left(\begin{array}{c} 512 \\ \sim \\ 512 \\ 256 \\ \mathbf{G} \end{array} \right) \cdot \begin{array}{c} \hline 1 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 1 \\ 0 \ 1 \ 0 \ 0 \\ \hline \end{array} \cdot \left(\begin{array}{c} 512 \\ 256 \\ \sim \\ \star \end{array} \right)}_{\text{ODT résultat miroir}}$$

Cette expression devient pour nous la forme standard d'une tâche ARRAY-OL :

$$\begin{pmatrix} M_o \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S_o \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_o \quad \mathcal{F}_o \quad \mathbf{0} | \cdot \begin{pmatrix} Q \\ D_o \\ D_r \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_r \quad \mathbf{0} \quad \mathcal{F}_r} \cdot \begin{pmatrix} M_r \\ \star \end{pmatrix}$$

Un autre avantage de cette forme est que si une autre tâche travaille avec, comme opérande, le tableau résultat de la première, on peut alors composer les deux ODT puisque le tableau en question fait le lien.

$$\text{ODT}_{\text{tâche}_1} \dots \begin{pmatrix} M_r \\ \star \end{pmatrix} \cdot \begin{pmatrix} M_r \\ \mathbf{M} \end{pmatrix} \dots \text{ODT}_{\text{tâche}_2}$$

On peut ainsi définir par une expression une séquence de tâches liées par un seul tableau.

3.3.2 Forme ODT d'une hiérarchie

Soit une tâche supérieure

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}\mathcal{F}_1 | \cdot \begin{pmatrix} Q \\ D_1 \\ D_2 \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}\mathcal{F}_2} \cdot \begin{pmatrix} M_2 \\ \star \end{pmatrix}$$

appelant par hiérarchie une séquence de tâches

$$\begin{pmatrix} D_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}\mathcal{F}'_1 | \cdot \begin{pmatrix} Q' \\ D'_1 \\ D'_2 \\ \mathbf{G} \end{pmatrix} \dots \dots \begin{pmatrix} Q'' \\ D''_1 \\ D''_2 \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}\mathcal{F}''_2} \cdot \begin{pmatrix} D_2 \\ \star \end{pmatrix}$$

Les relations existantes entre la tâche et sa hiérarchie se situent au niveau du tableau opérande de la première sous-tâche et du tableau résultat de la dernière sous-tâche. Les points utilisés dans ces deux tableaux correspondent en fait aux indices des gabarits d'ajustage de la tâche supérieure (D_1 et D_2). En ramenant la hiérarchie au niveau supérieur (i.e. en remplaçant les gabarits d'ajustage par la sous-hiérarchie puis en les distribuant), on peut donc représenter l'ensemble par :

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}\mathcal{F}_1 | \cdot \left(\overline{\begin{pmatrix} D_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}\mathcal{F}'_1 | \cdot \begin{pmatrix} Q' \\ D'_1 \\ D'_2 \\ \mathbf{G} \end{pmatrix} \dots \dots \begin{pmatrix} Q'' \\ D''_1 \\ D''_2 \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}\mathcal{F}''_2} \cdot \begin{pmatrix} D_2 \\ \star \end{pmatrix}} \right) \cdot \overline{\mathcal{P}\mathcal{F}_2} \cdot \begin{pmatrix} M_2 \\ \star \end{pmatrix}$$

c'est-à-dire

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{PF}_1 | \cdot \begin{pmatrix} Q \\ D_1 \\ D_2 \\ \mathbf{M} \end{pmatrix} \cdot \begin{vmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathcal{PF}'_1 \\ \mathbf{0} & \mathbf{0} \end{vmatrix} \cdot \begin{pmatrix} Q \\ Q' \\ D'_1 \\ D'_2 \\ \mathbf{G} \end{pmatrix} \dots \begin{pmatrix} Q \\ Q'' \\ D''_1 \\ D''_2 \\ \mathbf{G} \end{pmatrix} \cdot \overline{\begin{vmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathcal{PF}''_2 \end{vmatrix}} \cdot \begin{pmatrix} Q \\ D_1 \\ D_2 \\ \star \end{pmatrix} \cdot \overline{\mathcal{PF}_2} \cdot \begin{pmatrix} M_2 \\ \star \end{pmatrix}$$

Inversement hiérarchiser une séquence de tâches consiste à trouver des dimensions de pavage identiques, elles seront factorisées pour former les dimensions de pavage de la tâche supérieure. Cette transformation ne suffit pas à garantir la validité de la hiérarchie, puisqu'on ampute les tâches d'une partie de leurs itérations, la caractéristique essentielle dans cette opération est donc d'assurer que les itérations résultantes sont cohérentes i.e. que les points engendrés par chaque tâche sont suffisants pour la suivante.

Soit la séquence de tâches suivantes dont on a réussi à séparer la partie de pavage en deux avec un gabarit commun :

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}_1^o \quad \mathcal{P}'_1^o \quad \mathcal{F}_1^o | \cdot \begin{pmatrix} Q \\ Q_1 \\ D_1 \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_1^r \quad \mathcal{P}'_1^r \quad \mathcal{F}_1^r} \dots \\ \dots | \mathcal{P}_n^o \quad \mathcal{P}'_n^o \quad \mathcal{F}_n^o | \cdot \begin{pmatrix} Q \\ Q_n \\ D_n \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_n^r \quad \mathcal{P}'_n^r \quad \mathcal{F}_n^r} \cdot \begin{pmatrix} M_{n+1} \\ \star \end{pmatrix}$$

La propriété de cohérence exprimée plus haut se traduit par ce test pour toutes les tâches :

$$\begin{pmatrix} M_{i+1} \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}_i^r \quad \mathcal{P}'_i^r \quad \mathcal{F}_i^r | \cdot \begin{pmatrix} Q \\ Q_i \\ D_i \\ \mathbf{G} \end{pmatrix} \blacktriangleleft \begin{pmatrix} \vec{q} \\ \mathbb{Z}^s \\ \mathbb{Z}^t \end{pmatrix} \subset \begin{pmatrix} M_{i+1} \\ \mathbf{M} \end{pmatrix} \cdot | \mathcal{P}_{i+1}^o \quad \mathcal{P}'_{i+1}^o \quad \mathcal{F}_{i+1}^o | \cdot \begin{pmatrix} Q \\ Q_{i+1} \\ D_{i+1} \\ \mathbf{G} \end{pmatrix} \blacktriangleleft \begin{pmatrix} \vec{q} \\ \mathbb{Z}^{s'} \\ \mathbb{Z}^{t'} \end{pmatrix}$$

On peut alors hiérarchiser ainsi :

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot \left| \begin{array}{ccccc} \mathcal{P}'_1 & \mathcal{P}'_1 & \mathcal{F}'_1 & \mathbf{0} & \mathbf{0} \end{array} \right| \cdot \begin{pmatrix} Q \\ Q_1 \\ D_1 \\ Q_n \\ D_n \\ \mathbf{G} \end{pmatrix} \left| \begin{array}{ccccc} \mathcal{P}'_n & \mathbf{0} & \mathbf{0} & \mathcal{P}'_n & \mathcal{F}'_n \end{array} \right| \cdot \begin{pmatrix} M_{n+1} \\ \star \end{pmatrix}$$

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot \left| \mathbf{1} \right| \cdot \begin{pmatrix} Q_1 \\ D_1 \\ \mathbf{G} \end{pmatrix} \frac{\overline{\mathcal{P}'_1} \ \overline{\mathcal{F}'_1}}{\dots} \left| \begin{array}{cc} \mathcal{P}'_n & \mathcal{F}'_n \end{array} \right| \cdot \begin{pmatrix} Q_n \\ D_n \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathbf{1}} \begin{pmatrix} M_{n+1} \\ \star \end{pmatrix}$$

3.3.3 Quelques propriétés sur les ODT

Nous listons ici quelques propriétés générales sur les ODT utiles dans la suite du développement.

1. Propriétés sur la composition des opérateurs modulo, éclatement et gabarit avec le même paramètre $\left(\begin{pmatrix} \vec{x} \\ \mathbf{M} \end{pmatrix}, \begin{pmatrix} \vec{x} \\ \star \end{pmatrix}, \begin{pmatrix} \vec{x} \\ \star\star \end{pmatrix}, \text{ et } \begin{pmatrix} \vec{x} \\ \mathbf{G} \end{pmatrix} \right)$:

$\diagdown \bullet$	\mathbf{M}	\star	$\star\star$	\mathbf{G}
\mathbf{M}	\mathbf{M}	\mathbf{G}	\mathbf{M}	\mathbf{G}
\star	$\star\star$	\star	$\star\star$	\star
$\star\star$	$\star\star$	\star	$\star\star$	\star
\mathbf{G}	\mathbf{M}	\mathbf{G}	\mathbf{M}	\mathbf{G}

2. Propriétés arithmétiques sur la projection et le décalage :

$$|\mathcal{M}_1| \cdot |\mathcal{M}_2| = |\mathcal{M}_1 \times \mathcal{M}_2|$$

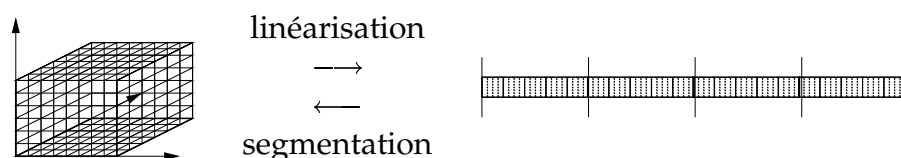
$$|\mathcal{M}_1| \cdot \begin{pmatrix} \mathcal{S} \\ \mathbf{S} \end{pmatrix} = \begin{pmatrix} \mathcal{M}_1 \times \mathcal{S} \\ \mathbf{S} \end{pmatrix} \cdot |\mathcal{M}_1|$$

3. Segmentation de gabarit.

Soit \vec{g} et $\vec{\lambda}$ deux vecteurs de même taille tels que $\forall i, \lambda_i \setminus g_i$ alors

$$\begin{pmatrix} g_1 \\ \vdots \\ g_n \\ \mathbf{G} \end{pmatrix} = \left| \begin{array}{cccc} \lambda_1 & & & \\ & \ddots & & \\ & & \lambda_n & \\ & & & \mathbf{1} \end{array} \right| \cdot \begin{pmatrix} g_1/\lambda_1 \\ \vdots \\ g_n/\lambda_n \\ \lambda_1 \\ \vdots \\ \lambda_n \\ \mathbf{G} \end{pmatrix} \cdot \frac{\lambda_1 \quad \mathbf{1}}{\dots \quad \lambda_n \quad \mathbf{1}}$$

Cette équivalence caractérise le fait de segmenter (dans le sens contraire de linéariser) des dimensions. Ainsi dans l'expression précédente chaque dimension est découpée en deux : la première représente le quotient de la division ; la seconde le reste.



4. Propriété sur les créations de dimensions.

Une segmentation $\overline{\mathcal{S}}$ est un « créateur de dimensions » si elle vérifie les propriétés suivantes : la matrice ne contient que des 1 et des 0, chaque ligne contient exactement un 1, chaque colonne contient au plus un 1¹. On impose que les dimensions créées se trouvent regroupées en fin de matrice². Ainsi \mathcal{S} est de la forme $(\mathbf{1} \ \mathbf{0})$

Le symétrique d'un tel ODT, qui est la projection $|\mathbf{1} \ \mathbf{0}|$, représente une réduction de dimensions. Les premières dimensions ressortent à l'identique et les dernières sont oubliées. La segmentation doit donc « recréer » ces dimensions. On peut la représenter par :

$$(x) \mapsto \left\{ \begin{pmatrix} x \\ \mathbb{Z}^n \end{pmatrix} \right\}$$

Il est possible de déplacer une segmentation « créateur de dimensions » dans une composition d'ODT suivant les règles ci-après :

¹Les colonnes non nulles ne modifient pas la dimension entrante ; les colonnes nulles créent de nouvelles dimensions.

²Sans perte de généralité ; à une simple renumérotation des dimensions près.

$$\begin{aligned}
\underline{\mathbf{1} \ \mathbf{0}} \cdot \begin{pmatrix} \vec{g} \\ \mathbf{G} \end{pmatrix} &= \begin{pmatrix} \vec{g} \\ \sim \\ \mathbf{G} \end{pmatrix} \cdot \underline{\mathbf{1} \ \mathbf{0}} \\
\underline{\mathbf{1} \ \mathbf{0}} \cdot |\mathcal{P}| &= \begin{vmatrix} \mathcal{P} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} \end{vmatrix} \cdot \underline{\mathbf{1} \ \mathbf{0}} \\
\underline{\mathbf{1} \ \mathbf{0}} \cdot \begin{pmatrix} \vec{m} \\ \mathbf{M} \end{pmatrix} &= \begin{pmatrix} \vec{m} \\ \sim \\ \mathbf{M} \end{pmatrix} \cdot \underline{\mathbf{1} \ \mathbf{0}} \\
\underline{\mathbf{1} \ \mathbf{0}} \cdot \begin{pmatrix} \vec{s} \\ \mathbf{S} \end{pmatrix} &= \begin{pmatrix} \vec{s} \\ \mathbf{0} \\ \mathbf{S} \end{pmatrix} \cdot \underline{\mathbf{1} \ \mathbf{0}} \\
\underline{\mathbf{1} \ \mathbf{0}} \cdot \begin{pmatrix} \vec{m} \\ \star \end{pmatrix} &= \begin{pmatrix} \vec{m} \\ \sim \\ \star \end{pmatrix} \cdot \underline{\mathbf{1} \ \mathbf{0}} \\
\underline{\mathbf{1} \ \mathbf{0}} \cdot \underline{\mathcal{P}} &= \underline{\mathcal{P} \ \mathbf{0}}
\end{aligned}$$

La dernière égalité résulte simplement de la symétrie de la propriété 2 sur la combinaison des projections.

3.3.4 ODT exact

Des propriétés du domaine ARRAY-OL pour la partie résultat des tâches décrites en section 2.2.5 découlent des propriétés sur la forme des ODT représentant les tâches ARRAY-OL.

Soit un ODT représentant une demi-tâche résultat ARRAY-OL. Cet ODT est de la forme

$$|\mathcal{P}| \cdot \begin{pmatrix} \vec{g} \\ \mathbf{G} \end{pmatrix}$$

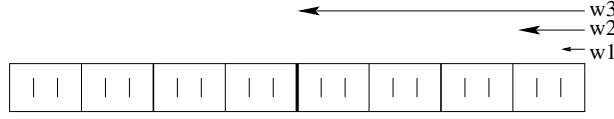
avec (g_i) les composantes de \vec{g} et (m_i) celles de \vec{m} .

On remarque déjà qu'il n'y a ni opérateur de décalage ni modulo. De plus les autres opérateurs vérifient certaines propriétés :

- chaque colonne de \mathcal{P} a au plus une valeur non nulle ;
- chaque ligne (i) de \mathcal{P} est telle que si on classe ses valeurs non nulles par ordre croissant $(w_{i,1} \dots w_{i,n})$ alors

$$\begin{aligned}
w_{i,1} &= 1, \\
\forall j, \quad w_{i,j} &\setminus w_{i,j+1}, \\
g_j &= w_{i,j+1}/w_{i,j}, \\
w_{i,n} \cdot g_n &= m_i
\end{aligned}$$

Ces caractéristiques signifient que l'ensemble des $(w_{i,*}, g_*)$ segmente exactement la $i^{\text{ème}}$ dimension.



Un ODT représentant une demi-tâche ARRAY-OL remplissant ces conditions est qualifié d'exact.

De plus, ces caractéristiques sont réciproques : si un ODT d'une demi-tâche résultat vérifient ces caractéristiques alors cette demi-tâche vérifient les bonnes spécifications ARRAY-OL.

3.3.5 Inversion d'un ODT exact

La segmentation d'une dimension décrite ci-dessus permet de retrouver simplement les itérations à partir du point.

$$x = |w_{i,1} \dots w_{i,n}| \cdot \begin{pmatrix} v_1 < g_1 \\ \vdots \\ v_n < g_n \end{pmatrix} \Rightarrow v_j = \frac{x}{w_{i,j}} \bmod g_{i,j} \Rightarrow \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} g_{i,*} \\ \mathbf{M} \end{pmatrix} \cdot \lfloor \begin{array}{c} \frac{1}{w_{i,1}} \\ \vdots \\ \frac{1}{w_{i,n}} \end{array} \rfloor \cdot (x)$$

En généralisant le procédé à toutes les dimensions, on peut obtenir alors une représentation par projection du miroir de l'ODT.

$$\left(| \mathcal{P} | \cdot \begin{pmatrix} \vec{g} \\ \mathbf{G} \end{pmatrix} \right)^{-1} = \begin{pmatrix} \vec{g} \\ \mathbf{M} \end{pmatrix} \cdot \lfloor \overline{\mathcal{P}''} \rfloor \cdot | \mathcal{P}' |$$

avec

- \mathcal{P}' obtenu à partir de \mathcal{P} par transposition et inversion des termes non nuls ;
- \mathcal{P}'' une matrice diagonale portant un 1 si la ligne correspondante de \mathcal{P}' est non nulle et 0 sinon.

La segmentation \mathcal{P}'' assure la création des dimensions. Il s'agit, en fait, de celles de l'ajustage opérande qui n'interviennent pas dans les matrices de pavage/ajustage résultat mais qu'on retrouve dans le gabarit central. Ces dimensions sont regroupées en

$$\text{fin de matrice : } \mathcal{P}'' = \begin{array}{cc} \mathbf{1} & 0 \\ 0 & 0 \end{array} .$$

3.3.6 Modulo redondant

Comme on le verra lors des transformations, on sera plusieurs fois bloqué par la présence d'un opérateur modulo qui empêchera la recombinaison d'autres opérateurs. Cette section présente quelques cas particuliers d'ODT où le modulo peut être éliminé de façon sûre.

► **Sur un ODT exact**

On reprend l'expression de l'inversion d'une projection exacte mais en négligeant les dimensions créées par la segmentation $\underline{\mathcal{P}''}$. On ne s'intéresse donc qu'à $\left(\begin{array}{c} \vec{g}_{/P'} \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}'}$

Pour une dimension m_i , on a $(q_{i,j} \times w_{i,j} \setminus m_i)$ donc $(\frac{m_i}{w_{i,j}} \bmod q_{i,j} = 0)$. Cela implique, en particulier, que si une valeur du vecteur d'entrée est un multiple de la taille du tableau alors les valeurs correspondantes du vecteur après la projection sont des multiples du gabarit. Cela implique, en outre, que si on remplace le gabarit final par un modulo le vecteur résultat sera nul :

$$\left(\begin{array}{c} \vec{g}_{/P'} \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}'} \triangleleft \begin{pmatrix} 0 \\ \vdots \\ m_i \\ \vdots \\ 0 \end{pmatrix} = \vec{0}$$

Cette propriété signifie que le modulo QD qui suit la projection effectue implicitement un modulo sur les dimensions avant la projection. On a donc l'équivalence suivante :

$$\left(\begin{array}{c} \vec{g}_{/P'} \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}'} \cdot \left(\begin{array}{c} \vec{m} \\ \mathbf{M} \end{array} \right) = \left(\begin{array}{c} \vec{g}_{/P'} \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}'}$$

Les dimensions créées par \mathcal{P}'' n'ayant pas de contraintes, un gabarit ou un modulo amèneront au même résultat. On conserve cette égalité en rajoutant $\underline{\mathcal{P}''}$.

► **Généralisation**

Soit l'ODT $\left(\begin{array}{c} M_2 \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}} \cdot \left(\begin{array}{c} M_1 \\ \mathbf{M} \end{array} \right)$. Si la projection est telle que tous les points d'entrée multiples de M_1 donnent des points résultats multiples de M_2 i.e.

$$\left(\begin{array}{c} M_2 \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}} \cdot \left(\begin{array}{c} M_1 \\ \star \end{array} \right) \triangleleft \vec{0} = \vec{0}$$

Alors on a :

$$\left(\begin{array}{c} M_2 \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}} \cdot \left(\begin{array}{c} M_1 \\ \mathbf{M} \end{array} \right) = \left(\begin{array}{c} M_2 \\ \mathbf{M} \end{array} \right) \cdot \lfloor \cdot \rfloor_{\mathcal{P}}$$

3.4 Transformation fondamentale : la fusion

Nous nous attaquons maintenant à l'opération qui sert de base au processus de transformation i.e. la fusion de deux tâches. Il s'agit de replacer une séquence de deux

tâches par une tâche hiérarchique.

Considérons deux tâches T_1 et T_2 . Pour simplifier, nous admettrons que T_1 consomme un seul tableau A_1 et produit un seul tableau A_2 ; que T_2 consomme A_2 et produit A_3 . Chaque tâche est exprimée par un ODT :

$$T_1 \rightarrow \begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S_1 \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_{op,1} | \cdot \begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_{res,1}}$$

$$T_2 \rightarrow \begin{pmatrix} M_2 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S_2 \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_{op,2} | \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_{res,2}}$$

avec M_i le modulo adéquat au tableau A_i ; $\mathcal{P}_{op,i}$ et $\mathcal{P}_{res,i}$ les vecteurs de pavage/ajustage pour l'opérande et le résultat; G_i^q le gabarit du pavage; G_i^d l'ensemble des gabarits d'ajustage opérande et résultat.

Une hiérarchisation consiste à fusionner les deux tâches T_1 et T_2 , c'est-à-dire à trouver une tâche permettant de passer directement du tableau A_1 au tableau A_3 . Celle-ci aura comme fonction une sous-application de deux tâches $T_{1'}$ et $T_{2'}$. Une telle hiérarchisation est illustrée par la figure 3.3.

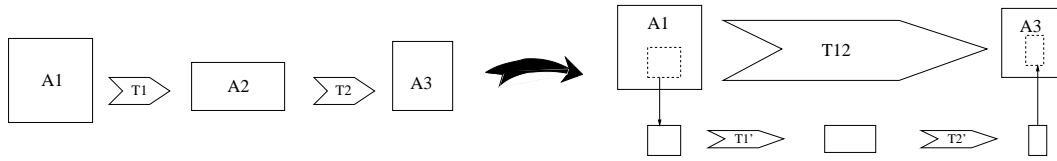


FIG. 3.3 – Fusion de deux tâches par hiérarchisation

La tâche T_{12} exprime les liens directs entre A_1 et A_3 . Du point de vue ODT, on peut l'obtenir en composant les tâches T_1 et T_2 :

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S_1 \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_{op,1} | \cdot \underbrace{\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_{res,1}} \cdot \begin{pmatrix} M_2 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S_2 \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_{op,2} | \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix}}_{\text{relation } QD_2 \rightarrow QD_1} \cdot \overline{\mathcal{P}_{res,2}}$$

L'étape essentielle de la hiérarchisation consiste à transformer cette expression en une forme d'ODT de tâche ARRAY-OL.

3.4.1 Orientation de transformation de l'expression des dépendances

Le sens naturel des dépendances dans ARRAY-OL est de déduire les points nécessaires des tableaux à partir des TE qu'on veut exécuter (cela se fait par simple multiplication matricielle alors que le contraire nécessiterait l'inversion de matrices entières). En conséquence, nous nous sommes concentrés sur les relations qui existent entre les ensembles de TE de la tâche antérieure et celui de la tâche finale (relation désignée par l'accolade dans l'expression précédente).

Notre travail principal va donc consister à transformer la partie de l'ODT concernant ces relations QD en un ODT de tâche ARRAY-OL mais travaillant, cette fois, entre les domaines d'itérations QD_1 et QD_2 :

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} S \\ \mathbf{S} \end{pmatrix} \cdot | P_1 \quad A_1 | \cdot \begin{pmatrix} Q \\ D \\ \mathbf{G} \end{pmatrix} \cdot \overline{P_2 \quad A_2} \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \star \end{pmatrix}$$

L'itération sur D donne un motif opérande dans QD_1 et un motif résultat dans QD_2 . Ces deux motifs doivent vérifier deux contraintes pour que la fusion soit correcte : 1– Ils doivent correspondre à des motifs spatiaux complets (les dimensions G_1^d et G_2^d doivent être complètes) ; 2– Les itérations du motif sur QD_1 doivent engendrer sur le tableau intermédiaire assez de points pour pouvoir exécuter les itérations du motif de QD_2 (figure 3.4).

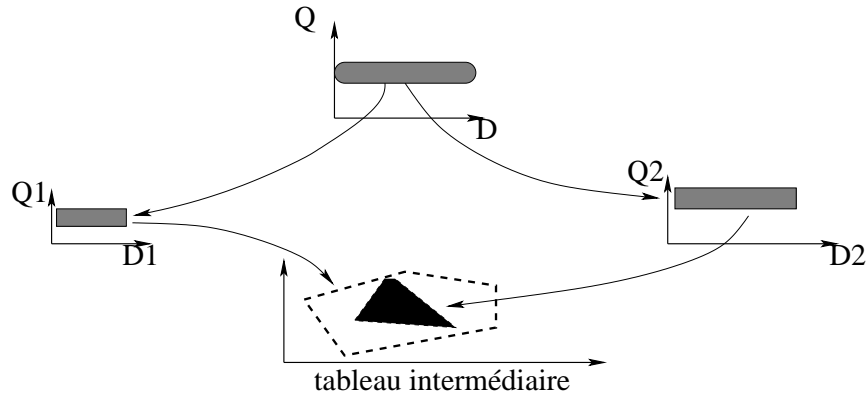


FIG. 3.4 – Passage de QD_1 à QD_2

Pour arriver à la forme voulue, nous avons deux directions possibles :

1. Trouver quelles itérations exécuter sur la seconde tâche avec un certain nombre d'itérations de la première. Ceci équivaut à inverser $| \mathcal{P}_{op,2} |$ (figure 3.5(a)) ;
2. Trouver les itérations nécessaires sur la première tâche pour exécuter un certain nombre d'itérations de la seconde. Ceci équivaut à inverser $\overline{\mathcal{P}_{res,1}}$ (figure 3.5(b)).

Nous choisissons cette seconde possibilité pour les raisons suivantes :

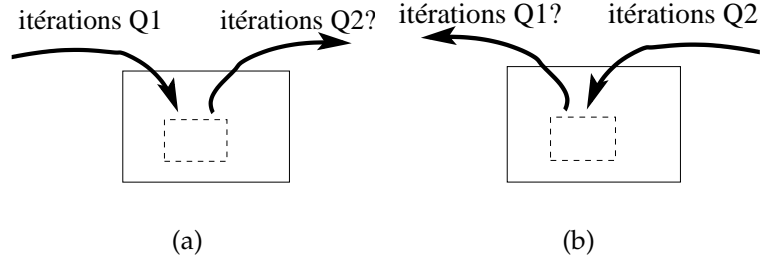


FIG. 3.5 – Deux directions pour identifier les dépendances QD

1. Sémantiquement, le remplissage du tableau résultat constitue le but des tâches alors que les valeurs des tableaux précédents sont seulement nécessaires.
2. Pratiquement, l'inversion d'un ODT exact (comme l'est $\overline{\mathcal{P}_{res,1}}$) est connu (section 3.3.5).

3.4.2 Dépendances fractionnaires

► Avec l'ODT inverse

- Partant d'un motif résultat de la tâche T_2 référencé par son indice de pavage $q \in G_2^q$:
- $\left(\begin{matrix} M_2 \\ \mathbf{M} \end{matrix} \right) \cdot \left(\begin{matrix} S_2 \\ \mathbf{S} \end{matrix} \right) \cdot | \mathcal{P}_{op,2} | \blacktriangleleft \left\{ \begin{pmatrix} q \\ x \end{pmatrix} ; \forall x \in G_2^d \right\}$ représente l'ensemble des points opérands nécessaires sur le tableau précédent (A_1).
 - Grâce à l'inversion de l'ODT résultat de la tâche T_1 sur ce même tableau, on a une expression de l'ensemble QD permettant de produire au moins tous les points opérands nécessaires :

$$\underbrace{\left(\begin{matrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{matrix} \right) \cdot \left[\begin{array}{c|c} \mathbf{1} & 0 \\ \hline 0 & 0 \end{array} \right] \cdot \left| \begin{matrix} {}^t \mathcal{P}'_{res,1} \\ {}^t \mathcal{F}'_{res,1} \end{matrix} \right|}_{\text{ODT miroir}} \cdot \left(\begin{matrix} M_2 \\ \mathbf{M} \end{matrix} \right) \cdot \left(\begin{matrix} S_2 \\ \mathbf{S} \end{matrix} \right) \cdot | \mathcal{P}_{op,2} | \blacktriangleleft \left\{ \begin{pmatrix} q \\ x \end{pmatrix} ; \forall x \in G_2^d \right\}$$

- En intégrant à l'ODT symétrique résultat, l'opérateur de modulo sur M_2 provenant de la tâche opérande, on peut alors appliquer la propriété des ODT exacts exposées plus haut (3.3.6) pour finalement supprimer ce modulo et transformer le gabarit de gauche. L'expression devient alors :

$$\left(\begin{matrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{matrix} \right) \cdot \left[\begin{array}{c|c} \mathbf{1} & 0 \\ \hline 0 & 0 \end{array} \right] \cdot \left| \begin{matrix} {}^t \mathcal{P}'_{res,1} \\ {}^t \mathcal{F}'_{res,1} \end{matrix} \right| \cdot \left(\begin{matrix} S_2 \\ \mathbf{S} \end{matrix} \right) \cdot | \mathcal{P}_{op,n} | \blacktriangleleft \left\{ \begin{pmatrix} q \\ x \end{pmatrix} ; \forall x \in G_2^d \right\}$$

- Cela permet de déplacer l'OE de décalage vers la gauche en le multipliant par la matrice (3.3.3, propriété 2) et de fusionner les deux matrices $\mathcal{P}'_{res,1}$ et $\mathcal{P}_{op,2}$ et d'obtenir :

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \begin{pmatrix} {}^t\mathcal{P}'_{res,1} \times S_2 \\ {}^t\mathcal{F}'_{res,1} \times S_2 \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{matrix} {}^t\mathcal{P}'_{res,1} \times \mathcal{P}_{op,2} \\ {}^t\mathcal{F}'_{res,1} \times \mathcal{P}_{op,2} \end{matrix} \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix}$$

- Pour simplifier l'écriture, on renomme les valeurs issues des multiplications matricielles :

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \begin{pmatrix} S_p \\ S_f \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{matrix} \mathcal{P}'' \\ \mathcal{F}'' \end{matrix} \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix} \blacktriangleleft \left\{ \begin{pmatrix} q \\ x \end{pmatrix} ; \forall x \in G_2^d \right\}$$

► Complétion des motifs

On ne veut que des motifs entiers, il donc faut compléter les dimensions G_1^d de l'expression

- Ajout de l'ODT de complétion.

$$\begin{aligned} & \begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \begin{pmatrix} S_p \\ S_f \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{matrix} \mathcal{P}'' \\ \mathcal{F}'' \end{matrix} \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{M} \end{pmatrix} \\ & \subset \underbrace{\begin{pmatrix} \tilde{} \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \left| \mathbf{1} \ \mathbf{0} \right| \cdot \begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix}}_{\text{ODT de complétion}} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \begin{pmatrix} S_p \\ S_f \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{matrix} \mathcal{P}'' \\ \mathcal{F}'' \end{matrix} \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix} \end{aligned}$$

La projection élimine les dimensions partielles, la segmentation les recrée entièrement pleines (infinies) et le modulo final les limite aux bornes de l'ajustage.

- Pour revenir à une forme standard, on déplace la projection vers la droite et on la fusionne avec la projection fractionnaire (i.e. on élimine les anciennes dimensions d'ajustage)

$$\begin{pmatrix} \tilde{} \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\underline{\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}}} \cdot \begin{pmatrix} G_1^q \\ \mathbf{M} \end{pmatrix} \cdot \left| \begin{pmatrix} S_p \\ \mathbf{S} \end{pmatrix} \right| \cdot \left| \mathcal{P}'' \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix}$$

Notons que la segmentation qui suivait le décalage a disparu : elle n'était là que pour créer l'ajustage opérande qui vient de disparaître.

- Enfin, on déplace l'ODT formé de la segmentation et du gabarit vers la droite. Cela crée les dimensions d'ajustage contraintes pour les modulus et les gabarits, et des

opérateurs neutres pour les décalages et les projections.

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \underline{\cdot} \cdot \begin{pmatrix} \mathcal{S}_p \\ \mathbf{0} \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{array}{cc} \mathcal{P}'' & \mathbf{0} \\ \mathbf{0} & \mathbf{1} \end{array} \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \underline{\overline{\mathbf{1} \ \mathbf{0}}}$$

La segmentation n'a pas pu fusionner, on la retrouve donc tout à droite de l'expression.

- Les parties d'ajustage spatiales étant maintenant complètes, on va s'intéresser aux parties concernant le macro-pavage et le macro-ajustage i.e.

$$\begin{pmatrix} G_1^q \\ \mathbf{M} \end{pmatrix} \cdot \underline{\cdot} \cdot \begin{pmatrix} \mathcal{S}_p \\ \mathbf{S} \end{pmatrix} \cdot \left| \mathcal{P}'' \right| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix}$$

► Exemple numérique

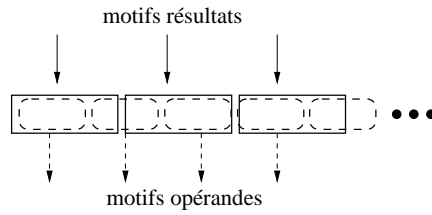


FIG. 3.6 – Recouvrement des motifs résultats et opérandes

L'ODT initial des relations *QD* illustré par la figure 3.6 est le suivant (*x* et *y* remplacent les bornes d'ajustages croisées qui n'interviennent pas dans les calculs) :

$$\begin{pmatrix} \sim \\ 3 \\ x \\ \mathbf{G} \end{pmatrix} \cdot \underline{\overline{\mathbf{3} \ 1 \ 0}} \cdot \begin{pmatrix} \sim \\ \star \end{pmatrix} \cdot \begin{pmatrix} \sim \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} 4 \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{array}{ccc} 2 & 0 & 1 \end{array} \right| \cdot \begin{pmatrix} \sim \\ y \\ 2 \\ \mathbf{G} \end{pmatrix}$$

Après l'inversion du résultat :

$$\begin{pmatrix} \sim \\ 3 \\ x \\ \mathbf{M} \end{pmatrix} \cdot \underline{\overline{\mathbf{1} \ 0 \ 0} \ \underline{\overline{\mathbf{0} \ 1 \ 0}}} \cdot \begin{pmatrix} 4/3 \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{array}{ccc} 2/3 & 0 & 1/3 \\ 2 & 0 & 1 \end{array} \right| \cdot \begin{pmatrix} \sim \\ y \\ 2 \\ \mathbf{G} \end{pmatrix}$$

Après complétion des motifs :

$$\left(\begin{array}{c} \sim = G_1^q \\ 3 \\ x \\ \mathbf{M} \end{array} \right) \cdot \left(\begin{array}{c} 4/3 = S_p \\ 0 \\ 0 \\ \mathbf{S} \end{array} \right) \cdot \left| \begin{array}{ccc|cc} 2/3 & 0 & 1/3 = \mathcal{P}'' & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array} \right| \cdot \left(\begin{array}{c} \sim \\ x \\ 2 \\ 3 \\ x \\ \mathbf{G} \end{array} \right) \cdot \begin{array}{|ccc|cc} \hline 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Sur cette dernière expression, on a séparé les blocs correspondant aux variables de l'expression théorique et on en a noté quelques unes.

3.4.3 Pavage entier

Sur l'expression précédente, on va visualiser les influences de G_2^q et G_2^d en dissociant la projection ($\mathcal{P}'' = \mathcal{M}_q \mathcal{M}_d$) :

$$\left(\begin{array}{c} G_1^q \\ \mathbf{M} \end{array} \right) \cdot \left(\begin{array}{c} S_p \\ \mathbf{S} \end{array} \right) \cdot \left| \mathcal{M}_q \quad \mathcal{M}_d \right| \cdot \left(\begin{array}{c} G_2^q \\ G_2^d \\ \mathbf{G} \end{array} \right)$$

Cette expression peut se décrire comme une tâche ARRAY-OL entre les espaces d'itérations Q_1 et Q_2 . Les macro-itérations sur G_2^d donnent de part et d'autre un macro-motif d'itérations opérandes et un macro-motif d'itérations résultats ; les macro-itérations sur G_2^q pavent ces macro-motifs sur les espaces d'itérations.

La figure 3.7 donne l'exemple du macro-motif origine (en grisé clair) et de celui décalé par un vecteur de pavage (en grisé foncé) pour les parties opérande (Q_1) et résultat (Q_2). La partie résultat est triviale puisque Q_2 est généré point par point. Par contre sur Q_1 , on peut non seulement avoir les points du macro-motif non entiers mais également les vecteurs de pavage non entiers.

On va commencer par revenir sur des points d'origines de motif entiers (i.e. sur des vecteurs de pavage entiers). Pour cela, on propose de regrouper les motifs sur chaque dimension de pavage (i.e. de prendre des multiples des vecteurs de pavage). Il suffira ensuite de calculer la partie entière du macro-motif d'origine (celui de la première itération) pour en déduire tous les autres (cf. la figure 3.8).

3.4.4 Regroupement des motifs

De façon pratique on veut appliquer des facteurs multiplicatifs (λ_i) sur chaque colonne de la matrice de pavage :

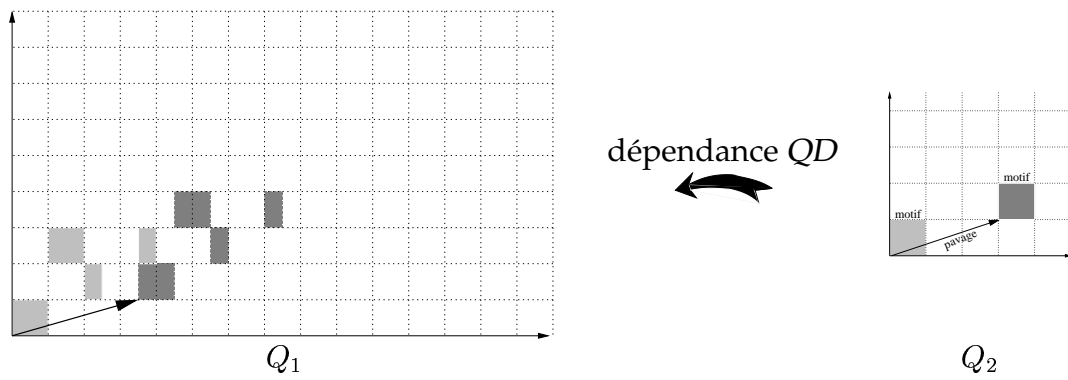


FIG. 3.7 – Vecteur de pavage non entier (Q_1)

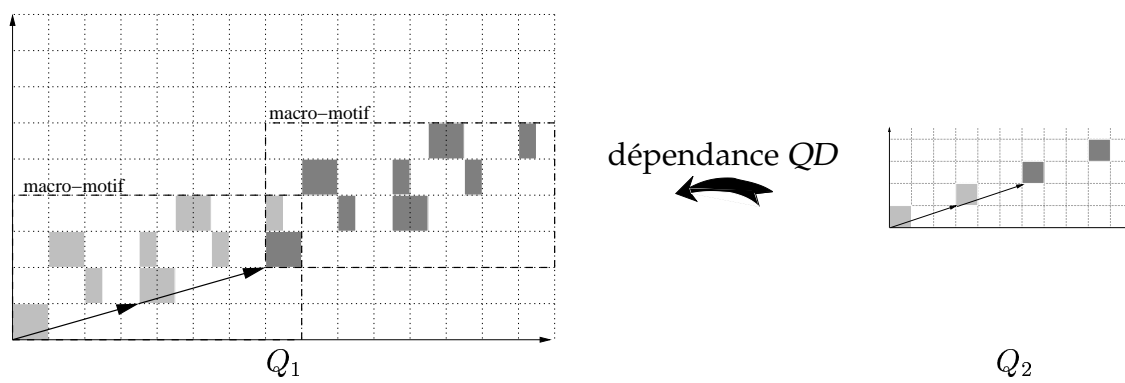


FIG. 3.8 – Regroupement de deux motifs sur cette dimension de pavage

$$\begin{array}{ccc} \lambda_1 & \dots & \lambda_r \\ \downarrow & & \downarrow \\ \boxed{\mathcal{P}^{q,q}} & & \end{array} \quad \Lambda = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_r \end{pmatrix}$$

Ces facteurs sont produits par la segmentation du gabarit G_2^q (cf. 3.3.3, propriété 3) : on rassemble Λ_i itérations de pavage sur chaque dimension

$$\begin{aligned} |\mathcal{M}_Q \quad \mathcal{M}_D| \cdot \begin{pmatrix} G_2^q \\ G_2^d \\ \mathbf{G} \end{pmatrix} &= |\mathcal{M}_Q \quad \mathcal{M}_D| \cdot \begin{vmatrix} \Lambda & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{vmatrix} \cdot \begin{pmatrix} G_2^q/\Lambda \\ \Lambda \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \begin{vmatrix} \Lambda & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{vmatrix} \\ &= |\mathcal{M}_Q \times \Lambda \quad \mathcal{M}_Q \quad \mathcal{M}_D| \cdot \begin{pmatrix} G_2^q/\Lambda \\ \Lambda \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \begin{vmatrix} \Lambda & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{vmatrix} \\ &= |\mathcal{M}'_Q \quad \mathcal{M}_Q \quad \mathcal{M}_D| \cdot \begin{pmatrix} G'^Q \\ \Lambda \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \begin{vmatrix} \Lambda & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{vmatrix} \end{aligned}$$

Ainsi $G'^Q = G_2^q/\Lambda$ représente le nouveau gabarit de pavage alors que le gabarit Λ passe dans la partie de macro-ajustage.

Remarque Si le nombre d'itérations n'est pas un multiple de λ_i , on ne peut segmenter la dimension. Dans ce cas on passe la dimension entière dans la partie d'ajustage, c'est-à-dire qu'on considère que λ_i est égal au nombre d'itérations.

3.4.5 Ajustage et décalage entier

► Découplage de l'ajustage

Les itérations du macro-pavage sont exécutées indépendamment les unes des autres, il existe donc une dépendance points à points entre les parties opérande et résultat. Par contre, les itérations d'ajustage forment un tout, ce qui signifie qu'il s'agit de dépendance entre des ensembles de liens (on n'est pas obligé de conserver l'ordre de production des points du moment qu'on les atteint tous). C'est pourquoi on se permet de dupliquer et de découpler les dimensions d'ajustage entre les deux côtés de l'expression :

$$| \mathcal{M}'_Q \quad \mathcal{M}_Q \quad \mathcal{M}_D \quad \mathbf{0} \quad \mathbf{0} | \cdot \left(\begin{array}{c} G'^Q \\ \left. \begin{array}{l} \Lambda \\ G_2^d \end{array} \right\} \text{macro-ajustage opérande} \\ \left. \begin{array}{l} \Lambda \\ G_2^d \end{array} \right\} \text{macro-ajustage résultat} \\ \mathbf{G} \end{array} \right) \cdot \begin{array}{ccccc} \hline \Lambda & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

L'intérêt de cette opération réside dans le fait qu'on peut maintenant modifier les deux parties indépendamment l'une de l'autre. En particulier, on va s'intéresser uniquement à la partie opérande, tout en gardant inchangée la partie résultat qui a déjà une forme standard entière.

On travaille donc dans la suite avec la partie de l'ajustage opérande en reprenant le décalage \mathcal{S}_p que nous avons mis de côté et qui n'est toujours pas entier :

$$\lfloor \cdot \left(\begin{array}{c} \mathcal{S}_p \\ \mathbf{S} \end{array} \right) \rfloor \cdot | \mathcal{M}_Q \quad \mathcal{M}_D | \cdot \left(\begin{array}{c} \Lambda \\ G_2^d \\ \mathbf{G} \end{array} \right)$$

► Pré-transformations

Ces opérations de pré-transformations n'éliminent pas les fractions mais réduisent leur influence :

1. La segmentation de l'ajustage (similaire à celle du pavage)

$$| v | \cdot \left(\begin{array}{c} q \\ \mathbf{G} \end{array} \right) = | \Lambda' v \quad v | \cdot \left(\begin{array}{c} q/\Lambda' \\ \Lambda' \\ \mathbf{G} \end{array} \right)$$

- permet de réduire la borne d'itération,
- peut agrandir la borne d'ajustage (i.e. ajouter des points) si celle-ci ne divise pas exactement Λ ;

2. La séparation de la partie fractionnaire :

$$| v | \cdot \left(\begin{array}{c} q \\ \mathbf{G} \end{array} \right) \subset | \text{Ent}(v) \quad \text{Frac}(v) | \cdot \left(\begin{array}{c} q \\ q \\ \mathbf{G} \end{array} \right)$$

augmente le nombre d'itérations³ (de q à $q \times q$) mais réduit la taille des vecteurs fractionnaires.

³D'où l'intérêt d'avoir réduit cette borne précédemment.

► **Ensemble englobant**

On va maintenant éliminer toutes les parties fractionnaires (les vecteurs d'ajustage mais aussi le décalage) en prenant l'englobant entier des points atteints par les vecteurs fractionnaires (rassemblés dans la matrice \mathcal{Frac}) :

$$\begin{pmatrix} \mathcal{S}_p \\ \mathbf{S} \end{pmatrix} \cdot |\mathcal{Frac}| \cdot \begin{pmatrix} q \\ \mathbf{G} \end{pmatrix} \subset \begin{pmatrix} (\text{borne inf}) \\ \mathbf{S} \end{pmatrix} \cdot |\mathbf{1}| \cdot \begin{pmatrix} (\text{borne sup}) - (\text{borne inf}) \\ \mathbf{G} \end{pmatrix}$$

On englobe donc les parties fractionnaires par l'ensemble de tous les points entiers entre les points (borne inf) et (borne sup).

Remarque Pour des vecteurs mono-dimensionnels, l'inclusion ci-dessus est en fait une égalité i.e. l'englobant n'ajoute aucun point inutile. Cela vient du fait qu'à ce moment là toutes les valeurs sont inférieures à 1.

► **Forme finale**

$$\lfloor \cdot \begin{pmatrix} \mathcal{S}' \\ \mathbf{S} \end{pmatrix} \cdot |\mathcal{M}'_D| \cdot \begin{pmatrix} G'^D \\ \mathbf{G} \end{pmatrix}$$

À ce stade tous les opérateurs de l'ODT sont de nouveau entiers et l'opérateur d'arrondi devient alors inutile.

3.4.6 Dépendances finales

L'expression entière des dépendances devient :

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{S}' \\ \mathbf{0} \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{array}{ccc} \mathcal{M}'_Q & \mathcal{M}'_D & \mathbf{0} \ \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \ \mathbf{1} \end{array} \right| \cdot \begin{pmatrix} G'^Q \\ G'^D \\ \Lambda \\ G_1^d \\ \mathbf{G} \end{pmatrix} \cdot \frac{\begin{array}{cccc} \Lambda & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}}{\hline}$$

Les transformations appliquées ont pu entraîner l'apparition de points redondants. On va donc appliquer une opération de simplification :

- cette simplification s'applique à un couple \vec{u}, \vec{v} de vecteurs colinéaires ($\alpha \vec{u} = \beta \vec{v} = \vec{w}$) associés aux itérateurs q_u et q_v ;
- on suppose que α et β sont premiers entre eux, quitte à les diviser par leur PGCD ;
- les seules points communs sont les multiples de \vec{w}

– on segmente l'une des deux dimensions pour faire apparaître \vec{w} (par exemple \vec{u}) :

$$|\vec{u} \quad \vec{v}| \cdot \begin{pmatrix} q_u \\ q_v \\ \mathbf{G} \end{pmatrix} = |\vec{w} \quad \vec{u} \quad \vec{v}| \cdot \begin{pmatrix} q_u/\alpha \\ \alpha \\ q_v \\ \mathbf{G} \end{pmatrix}$$

– le vecteur \vec{w} étant multiple de \vec{v} , si l'itérateur q_v dépasse l'indice de proportionnalité ($q_v \geq \beta$) alors on a des points communs. On les supprime en fusionnant \vec{w} avec \vec{v}

$$|\vec{u} \quad \vec{v}| \cdot \begin{pmatrix} \alpha \\ \beta(\frac{q_u}{\alpha} - 1) + q_v \\ \mathbf{G} \end{pmatrix}$$

Cette simplification va intervenir sur la matrice d'ajustage opérante, $\mathcal{M}'_D \cdot \begin{pmatrix} G'^D \\ \mathbf{G} \end{pmatrix}$, et simultanément sur les matrices de pavage opérante et résultat, $|\mathcal{M}'_Q| \cdot \begin{pmatrix} G'^Q \\ \mathbf{G} \end{pmatrix} \cdot \underline{\Lambda}$ afin de garder la cohérence point à point du pavage (en pratique, les deux matrices de pavage sont concaténées en $\begin{vmatrix} \mathcal{M}'_Q \\ \Lambda \end{vmatrix}$ pendant la simplification puis re-éclatées).

Finalement, nous atteignons la forme recherchée au départ :

$$\begin{pmatrix} G_1^q \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{S} \\ \mathbf{S} \end{pmatrix} \cdot |\mathcal{P}_1 \quad \mathcal{A}_1| \cdot \begin{pmatrix} Q \\ D \\ \mathbf{G} \end{pmatrix} \cdot \underline{\mathcal{P}_2 \quad \mathcal{A}_2}$$

3.4.7 Mise en forme hiérarchique

► Élimination du modulo

Ayant normalisé la forme ODT des dépendances d'itérations, nous pouvons revenir aux domaines spatiaux.

$$\begin{array}{c}
 \underbrace{\left(\begin{array}{c} G_1^q \\ G_1^d \\ \mathbf{M} \end{array} \right) \cdot \left(\begin{array}{c} \mathcal{S} \\ \mathbf{S} \end{array} \right) \cdot | \mathcal{P}_1 \quad \mathcal{A}_1 | \cdot \left(\begin{array}{c} G^Q \\ G^D \\ \mathbf{G} \end{array} \right) \cdot \overline{\mathcal{P}_2 \quad \mathcal{A}_2}} \\
 \swarrow \\
 \left(\begin{array}{c} M_1 \\ \mathbf{M} \end{array} \right) \cdot \left(\begin{array}{c} \mathcal{S}_1 \\ \mathbf{S} \end{array} \right) \cdot | \mathcal{P}_{op,1} | \leftarrow QD_1 \rightarrow \overline{\mathcal{P}_{res,1}} \quad \searrow \\
 \left(\begin{array}{c} M_2 \\ \mathbf{M} \end{array} \right) \cdot \left(\begin{array}{c} \mathcal{S}_2 \\ \mathbf{S} \end{array} \right) \cdot | \mathcal{P}_{op,2} | \leftarrow QD_2 \rightarrow \overline{\mathcal{P}_{res,2}}
 \end{array}$$

La tâche supérieure est représentée par la composition de 3 ODT : la demi-tâche opérande sur M_1 , l'ODT des dépendances d'itérations et l'ODT de demi-tâche résultat sur M_3 . Les deux sous-tâches sont spécifiées par l'ODT des tâches originales où vient s'insérer la partie d'ajustage opérande ou résultat des dépendances d'itérations.

On remarque que le modulo $\left(\begin{array}{c} G_1^q \\ G_1^d \\ \mathbf{M} \end{array} \right)$ séparant les projections constitue le dernier obstacle à la fusion. Nous avons identifié deux cas qui permettent d'éliminer purement et simplement ce modulo sans autre opération :

- le premier concerne son utilité du point de vue des contraintes : si les bornes atteintes par la partie opérande des dépendances d'itérations sont inférieures aux modulus ;
- le second est lié aux propriétés arithmétiques : si le modulo est redondant avec les modulus sur les tableaux (M_1 et M_2) (cf. 3.3.6)

Sinon une dernière transformation est nécessaire. Elle consiste créer un nouveau vecteur d'ajustage chargé de décrire entièrement la dimension gênante, cela peut contraindre aussi à passer certains vecteurs de pavage dans l'ajustage.

► Décalage d'origine des macro-motifs

Le macro-pavage a servi dans la tâche supérieure à déterminer l'origine des macro-motifs. Mais les deux sous-tâches vont travailler seulement sur la partie de macro-ajustage. Cela implique, en particulier, que les différents macro-motifs opérandes et

résultats du tableau intermédiaire doivent avoir une position relative fixe : l'écart des points d'origine ne doit pas dépendre du macro-pavage.

Si c'est le cas⁴, nous forçons la(es) dimension(s) de macro-pavage incriminée(s) à passer dans l'ajustage.

► Forme définitive

Il ne reste plus qu'à formater les expressions des trois tâches ARRAY-OL :

– la tâche supérieure

$$\begin{pmatrix} M_1 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{S}_1 + \mathcal{P}_{op,1} \times \mathcal{S} \\ \mathbf{S} \end{pmatrix} \cdot \left| \begin{array}{cc} \mathcal{P}_{op,1} \times \mathcal{P}_1 & \mathcal{P}_{op,1} \times \mathcal{A}_1 \end{array} \right| \cdot \begin{pmatrix} G^Q \\ G^D \\ \mathbf{G} \end{pmatrix} \\ \cdot \frac{\mathcal{P}_{res,2} \times \mathcal{P}_2 \quad \mathcal{P}_{res,2} \times \mathcal{A}_2}{\quad}$$

– la première sous-tâche

$$\begin{pmatrix} G'^D \\ G_1^d \\ \mathbf{M} \end{pmatrix} \cdot \left| \mathbf{1} \right| \cdot \begin{pmatrix} G'^D \\ G_1^d \\ \mathbf{G} \end{pmatrix} \cdot \frac{\mathcal{P}_{res,1} \times \mathcal{A}_1}{\quad} \cdot \begin{pmatrix} M_2 \\ \star \end{pmatrix}$$

– la seconde sous-tâche

$$\begin{pmatrix} M_2 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{S}_2 \\ \mathbf{S} \end{pmatrix} \cdot \left| \mathcal{P}_{op,2} \times \mathcal{A}_2 \right| \cdot \begin{pmatrix} G^D \\ G_2^d \\ \mathbf{G} \end{pmatrix} \cdot \frac{\mathbf{1}}{\quad} \cdot \begin{pmatrix} G^D \\ G_2^d \\ \star \end{pmatrix}$$

Remarque

- La présence des matrices identités dans les sous-tâches s'explique par le fait que l'ajustage des macro-motifs est effectué par la tâche supérieure. Ainsi les sous-tâches travaillent sur des macro-motifs bien formés.
- On peut facilement vérifier que la tâche supérieure et la seconde sous-tâche sont exactes, ce qui n'est pas forcément le cas de la première sous-tâche.

3.4.8 Cas d'une séquence de tâches quelconque

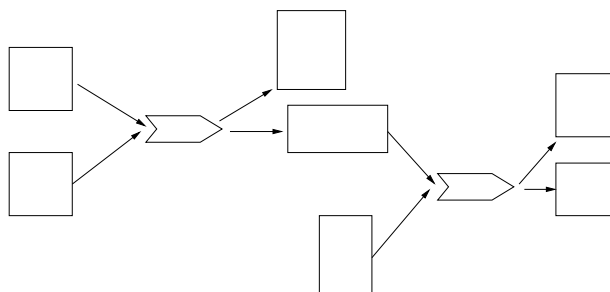
Nous avons traité le cas où les deux tâches à fusionner n'avaient chacune qu'un tableau opérande et qu'un tableau résultat. Dans la pratique les tâches ont souvent plusieurs tableaux opérandes (la tâche `FORMATION_DE_VOIES` de la VBL, par exemple, consomme un tableau de coefficients en plus du tableau des FFT), voir même, plusieurs

⁴Voir par exemple le point traitant du « corner-turn » de la section 3.6.2

tableaux résultats (par exemple, le tableau de sortie classique et un tableau de visualisation de résultats intermédiaires). Nous allons voir comment certains de ces cas peuvent être traités.

► Un seul tableau entre les deux tâches

On reste dans le cas où il n'y a qu'un seul tableau qui relie les deux tâches mais on peut avoir plusieurs tableaux opérands et résultats sur chaque tâche.

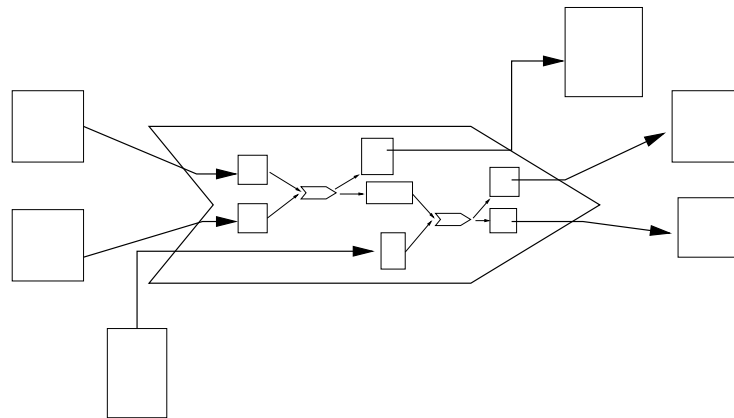


Dans ce cas, la première partie de l'algorithme, i.e. la transformation de la relation QD , est conservée puisqu'elle ne dépend que des caractéristiques agissant sur l'unique tableau de liaison.

Ensuite la reconstruction des expressions finales, en revenant au domaine spatiale, se fait en combinant tous les tableaux opérands puis résultats en un seul. Ceci est obtenu en concaténant les dimensions des tableaux ainsi que les matrices de pavage/ajustage. L'expression suivante montre un regroupement de deux demi-tâches :

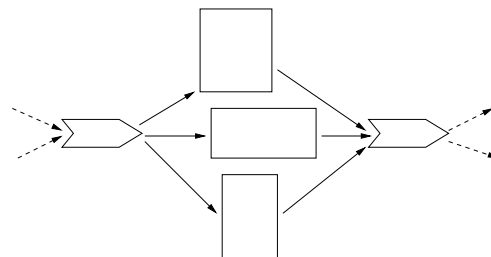
$$\left(\begin{array}{c} M_1 \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{cc} \mathcal{P}_1 & \mathcal{F}_1 \end{array} \right| \cdot \left(\begin{array}{c} Q \\ D_1 \\ \mathbf{G} \end{array} \right) \left. \vphantom{\begin{array}{c} M_1 \\ \mathbf{M} \end{array}} \right\} \left(\begin{array}{c} M_1 \\ M_2 \\ \mathbf{M} \end{array} \right) \cdot \left| \begin{array}{ccc} \mathcal{P}_1 & \mathcal{F}_1 & \mathbf{0} \\ \mathcal{P}_2 & \mathbf{0} & \mathcal{F}_2 \end{array} \right| \cdot \left(\begin{array}{c} Q \\ D_1 \\ D_2 \\ \mathbf{G} \end{array} \right)$$

Après avoir effectué les multiplications matricielles et les simplifications sur cette unique expression, les matrices sont finalement re-éclatées pour retrouver les tableaux d'origine.

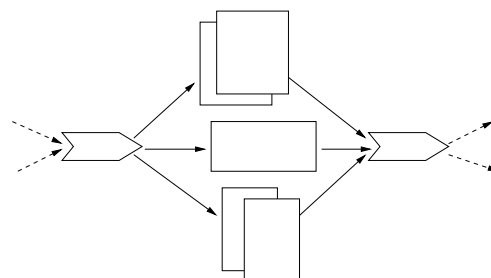


► Plusieurs tableaux entre deux tâches

Dans le cas où les tâches sont reliées par plusieurs tableaux, l'algorithme tel qu'il est décrit ne marche plus. En fait, on peut inverser les pavage/ajustage des tableaux de liaisons puisqu'ils sont exacts mais on se retrouve alors avec plusieurs expressions de relations *QD*. La relation exacte serait l'union de ces relations ce qui n'est *a priori* pas exprimable en ARRAY-OL.



L'alternative que nous avons est de choisir un des tableaux de liaisons pour la relation *QD* et de considérer les autres comme des tableaux résultats et opérandes distincts⁵. L'inconvénient est qu'il n'y a donc qu'un seul tableau qui bénéficie de la fusion, les autres ne pourront plus être transformés.



⁵distingués seulement dans l'algorithme de fusion mais au niveau des ressources mémoires cela reste des tableaux uniques

3.5 Vers une plate-forme de transformation

Les motivations de la transformation hiérarchique de code ont été expliquées au début du chapitre. Le procédé de fusion de deux tâches que nous venons de décrire en constitue l'opération unitaire et il est donc important d'en évaluer les conséquences. Cependant cette opération en elle-même n'est pas suffisante pour aboutir à des applications totalement hiérarchisées. Nous allons décrire dans la suite de la section, d'autres transformations dérivant de la fusion qui permettent de modifier les paramètres de celle-ci. Nous montrons comment on peut enchaîner de telles transformations. Le nombre de possibilités obtenues en fonction l'ordre d'application des transformations, nous amènera à nous interroger sur des critères objectifs qui pourraient nous orienter dans la bonne direction.

3.5.1 Évaluation de la fusion

Le but de la fusion est celui de la hiérarchisation c'est-à-dire introduire une transversalité dans l'exécution d'une chaîne de tâches. Cela permet de réduire les ressources temporaires ou améliorer la distribution de données mais cela se fait aux dépens d'autres facteurs comme la redondance des calculs.

► Réduction des ressources temporaires

Comme on vient de le voir, le tableau intermédiaire disparaît de l'application supérieure lors de la fusion. Cela signifie qu'il n'est utilisé que pour une exécution de la sous-application i.e. une fois l'exécution finie l'espace du tableau peut être réalloué pour l'exécution suivante. Donc l'espace réellement utile du tableau intermédiaire est celui de la sous-application.

En reprenant les notations précédentes, les 2 demi-tâches agissant sur ce tableau après fusion sont :

$$\begin{pmatrix} G^D \\ G_1^d \\ \mathbf{G} \end{pmatrix} \cdot \overline{\mathcal{P}_{res,1} \times \mathcal{A}_1} \cdot \begin{pmatrix} M_2 \\ \star \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} M_2 \\ \mathbf{M} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{S}_2 \\ \mathbf{S} \end{pmatrix} \cdot | \mathcal{P}_{op,2} \times \mathcal{A}_2 | \cdot \begin{pmatrix} G^D \\ G_2^d \\ \mathbf{G} \end{pmatrix}$$

On peut donc effectuer la mesure de l'encombrement (cf. 3.3.6) sur la partie résultat (puisqu'elle inclue, par construction, la partie opérande). La réduction finale se fait donc sur l'encombrement de chaque dimension i.e. la forme du tableau ne change pas.

On remarque alors qu'on peut perdre un certain avantage de la fusion si la forme des macro-motifs ne suit pas les dimensions ou si elle est très creuse (cf. la figure 3.9). Une fonction de réindexage du tableau pourrait être alors envisagée pour *redresser* ou compacter les motifs.

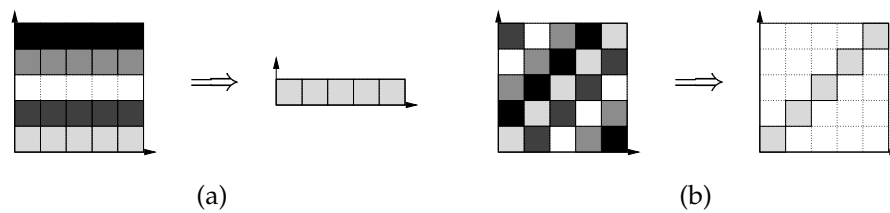


FIG. 3.9 – Illustration des problèmes de réduction du tableau temporaire. Sur la figure (a) les macro-motifs sont parallèles aux axes, le tableau se réduit exactement à la taille minimum d'un macro-motif. La figure (b) a les mêmes macro-motifs mais en diagonale, le tableau ne peut être réduit.

► Réduction de la latence

Une des conséquences de la hiérarchisation est que le tableau d'extrémité (le tableau résultat de la tâche supérieure) se remplit régulièrement à chaque méta-itération. Donc la latence sur la sortie est réduite à l'exécution de la sous-application i.e. à une partie du tableau intermédiaire.

On peut noter que cette réduction est proportionnelle au nombre de motifs qui forment le macro-motif et non à leur encombrement. Cela signifie, entre autre, qu'on peut avoir une mauvaise réduction de l'espace temporaire et une bonne réduction de la latence. Effectivement, sur l'exemple précédent, dans les deux cas la latence est réduite aux calculs de 5 motifs d'origine.

► Réduction des communication

On se place ici dans le contexte de mémoires distribuées. Sur l'application originale une méthode de distribution consiste à répartir un nombre égal de motifs de chaque tâche sur les différentes mémoires puisque ceux-ci sont indépendants. Cependant les motifs produits et consommés sur le tableau intermédiaire ne sont pas forcément identiques et bien agencés les uns aux autres. Sans une étude approfondie sur la stratégie de distribution, on est alors contraint d'effectuer une redistribution complète des données entre chaque tâche.

Pour l'éviter, il faut essayer de conserver la localité des motifs résultats et opérandes dans la distribution. La hiérarchisation peut nous aider dans ce cas puisqu'elle définit justement des macro-motifs permettant l'enchaînement des tâches. De manière simpliste, une tâche peut être distribuée idéalement mais pas une séquence de tâches. La hiérarchisation réduisant la séquence en une tâche, on retombe sur un cas simple de distribution.

► Violation des règles ARRAY-OL

On aura sans doute remarqué que lors de la fusion, le pavage/ajustage résultat du tableau intermédiaire n'est pas forcément exact au sens ARRAY-OL. En effet, il résulte de la dépendance arrière des motifs de la deuxième tâche. Il se peut donc que ses motifs ne soient pas parallèles aux axes, qu'ils ne remplissent pas le tableau⁶... enfin il se peut aussi qu'on calcul plusieurs fois les mêmes points.

Cette dernière entorse paraît évidemment la plus grave. Elle peut se produire, par exemple, au moment du calcul de l'ajustage entier. Le procédé de recombinaison de vecteurs décrit à la section 3.4.6 peut éliminer certaines redondances mais pas forcément toutes. Cela dit, cette violation des règles est atténuée par le fait que le recalcul des motifs provient de la re-exécution de la même TE. Ainsi les points calculés en double ont la même valeur et donc la tâche reste déterministe. Si on considère les règles strictes ARRAY-OL sur le résultat comme un manière d'obtenir un déterminisme lors de la spécification de l'application alors le processus de transformation conserve ce déterminisme. Notons qu'il existe un moyen d'éliminer toutes ces redondances ; il suffit de prendre un englobant des points qui soit parfaitement itérable⁷ mais cela se fait au prix de l'augmentation des motifs (et donc de calculs inutiles).

► Redondance des calculs

Comme on vient juste de la voir, il se peut que lors de la fusion, des redondances de calculs apparaissent non seulement dans la même exécution de la sous-application mais plus généralement entre deux exécutions. Ces dernières redondances ne viennent pas d'un problème d'itération comme précédemment mais du fait que deux macro-motifs résultats ont besoin chacun d'une partie d'un même motif opérande. On ne peut donc pas éliminer ces redondances aussi facilement.

Cette caractéristique constitue, selon nous, le facteur majeur qui limite les aspects bénéfiques (réduction d'espace, de latence...) de la hiérarchisation. C'est pourquoi ce phénomène est étudié plus en détail dans la section suivante (3.5.2).

► Limitations de la fusion

La fusion entre deux tâches consécutives n'est pas toujours possible :

- les deux tâches doivent n'avoir qu'un seul tableau commun. Dans le cas où il y en a plusieurs, on peut quand même fusionner mais seul un tableau bénéficiera de la fusion (cf. 3.4.8) ;
- la partie résultat de la première tâche doit être exacte. On peut arguer que cette contrainte fait partie d'ARRAY-OL et qu'elle se justifie par les caractéristiques clas-

⁶ce qui est heureux puisque c'était l'objectif. Plus embêtant est qu'il ne remplisse pas le tableau réduit

⁷ce qui est fait lors de la dernière opération de remise entière sur l'ajustage (cf. 3.4.5)

siques du traitement de signal. Cependant on pourrait songer à relâcher un peu cette contrainte. Tant qu'on peut trouver une forme linéaire de la segmentation (cf. 3.3.5) et éliminer le modulo du tableau (cf. 3.3.6), l'algorithme de fusion reste le même.

3.5.2 Recouvrement de motifs et de calculs

L'un des critères importants de qualité d'une fusion est la minimalisation des recouvrements, c'est-à-dire des recalculs. Nous allons identifier ici les différents cas de recouvrement et analyser les sources possibles de tels recouvrements.

► Définitions

Il existe deux degrés de recouvrement :

- le recouvrement spatial qui concerne deux motifs ayant certains points en communs ;
- le recouvrement de TE qui apparaît lorsqu'on exécute deux fois la même TE. D'un tel recouvrement découle évidemment un recouvrement spatial.

Ce deuxième type de recouvrement peut apparaître après une hiérarchisation entre deux méta-itérations. Il vient d'un recouvrement spatial dans l'ODT des dépendances QD utilisé pour la fusion, lui-même impliqué par un recouvrement des motifs opérands de la seconde tâche.

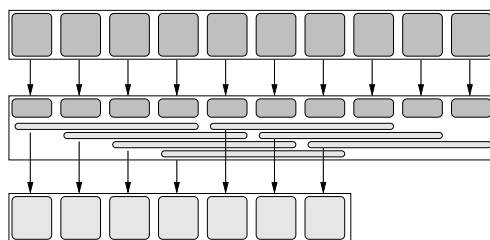


FIG. 3.10 – Exemple de recouvrement

Dans le cas de la figure 3.10, une fusion nous donnerait une hiérarchie ne produisant qu'un motif résultat. Le motif opérande correspondant a besoin de 4 motifs de la première tâche mais comme ces motifs opérands se recouvrent, les différentes exécutions de la sous-application vont recalculer 4 fois les motifs de la première tâche.

Au sein des recouvrements de TE, nous identifions deux types de recouvrements pour lesquels nous proposons une première prise en compte : le recouvrement itératif et le recouvrement cyclique.

► Recouvrement itératif

Dans l'exemple précédent, les recalculs se produisent pour des motifs consécutifs⁸. Cela signifie que si on agrandit le macro-motif suivant la dimension de pavage impliquant le recouvrement i.e. si la sous-application produit plusieurs motifs résultats en une fois, on diminue effectivement le nombre de recalculs puisqu'on va regrouper les motifs communs. Une suppression complète de ces recalculs passerait par l'intégration complète de la dimension dans le macro-motif.

Dans le cas précédent, cela reviendrait à annuler la hiérarchie. Pour la tâche de `FORMATION_DE_VOIES` de la VBL qui subit aussi ce genre de recouvrement, cela se traduirait par le regroupement des 512 hydrophones au lieu des 192 de départ. On triple alors la taille des tableaux intermédiaires mais on conserve la hiérarchisation sur l'axe temporel.

La réutilisation des points produits par les macro-motifs précédents dans le macro-motif courant (comme une sorte de résultat persistant à la sous-application) limiterait également ce recalcul, seules les TE non encore exécutées seraient déclenchées. Cependant une telle approche signifierait une modification profonde du schéma interne d'exécution ARRAY-OL.

► Recouvrement cyclique

Cette même tâche de `FORMATION_DE_VOIE` génère une autre sorte de recouvrement. Le motif de 192 hydrophones recouvre cycliquement la dimension : les derniers motifs qui débordent du tableau utilisent les points du début de la dimension (utilisation du modulo). Ceux-ci ont déjà été produits par les premières itérations. On nommera un tel recouvrement, un recouvrement cyclique.

La seule alternative pour éviter ce type de recalcul repose sur l'agrandissement du macro-motif sur toute la dimension du cycle (un agrandissement partiel ne servirait à rien).

Une réutilisation des points déjà calculés entraînerait une modification du schéma interne d'exécution encore plus importante que la précédente : il faudrait sauvegarder un ensemble de points intermédiaires pour les réutiliser en fin de tâche.

► Non-recouvrement

Un recouvrement de TE identifie deux appels identiques (mêmes opérandes, mêmes résultats) d'une même TE⁹. Un tel recouvrement peut être généré par une hiérarchisa-

⁸On suppose bien sûr que les motifs résultats sont produits dans l'ordre linéaire.

⁹Un non-recouvrement sur les TE n'implique pas un non-recouvrement de points alors que le contraire est vrai.

tion et une hiérarchisation ne produisant pas de recouvrement de TE assure qu'aucun calcul supplémentaire n'est effectué.

Un non-recouvrement de points sur les motifs (ou macro-motifs) traduit l'indépendance spatiale, en plus de fonctionnelle. À partir de cette remarque, il est facile d'imaginer une exécution parallèle sur mémoires distribuées (réseaux de stations Unix) des TE correspondantes.

3.5.3 Extension de la fusion élémentaire

Avec la fusion, nous avons défini une transformation de base d'une séquence de deux tâches en une tâche ARRAY-OL hiérarchique. Nous présentons maintenant les utilisations et extensions possibles de cette transformation élémentaire.

► Fusions chaînées

Une fois la fusion de deux tâches effectuée, on peut ignorer le fait que la tâche supérieure est hiérarchique. La nouvelle écriture de l'application est une séquence de tâches ; on peut à nouveau appliquer une fusion entre deux tâches consécutives. En particulier, la tâche résultant de la première fusion peut être impliquée dans cette nouvelle fusion (puisque'elle est exacte). On aboutit alors à une application à plusieurs niveaux de hiérarchie.

Différentes hiérarchies peuvent être construites selon l'ordre d'application des opérations de fusion de deux tâches (cf. la figure 3.11).

► Agrandissement du macro-motif

L'application d'une fusion produit des macro-motifs d'une taille minimale (dans le cadre de l'algorithme utilisé) ; on peut vouloir agrandir cette taille pour des raisons d'efficacité (cf. les problèmes de recouvrement vus en 3.5.2). En pratique, cette action est équivalente à la recherche du méta-pavage, il s'agit de choisir une segmentation du méta-pavage pour en reporter une partie dans le méta-ajustage.

On agrandit de $\Lambda = (\lambda_i)$ sur chaque dimension (figure 3.12) avant d'appliquer à nouveau les étapes successives de transformation pour reconstruire la hiérarchie.

Si on avait agrandi brutalement le macro-motif en concaténant les nouvelles dimensions aux anciennes (ce qui est nécessaire quand on applique l'agrandissement sur une hiérarchie d'origine), on aurait alors provoqué un agrandissement équivalent des TE à effectuer, du macro-motif et du tableau temporaire. Il est donc nécessaire de conserver les dépendances entre les macro-motifs (en fait, l'expression QD) pour que l'agrandissement tienne compte des recouvrements et puisse minimiser le nombre de TE et la taille du macro-motif.

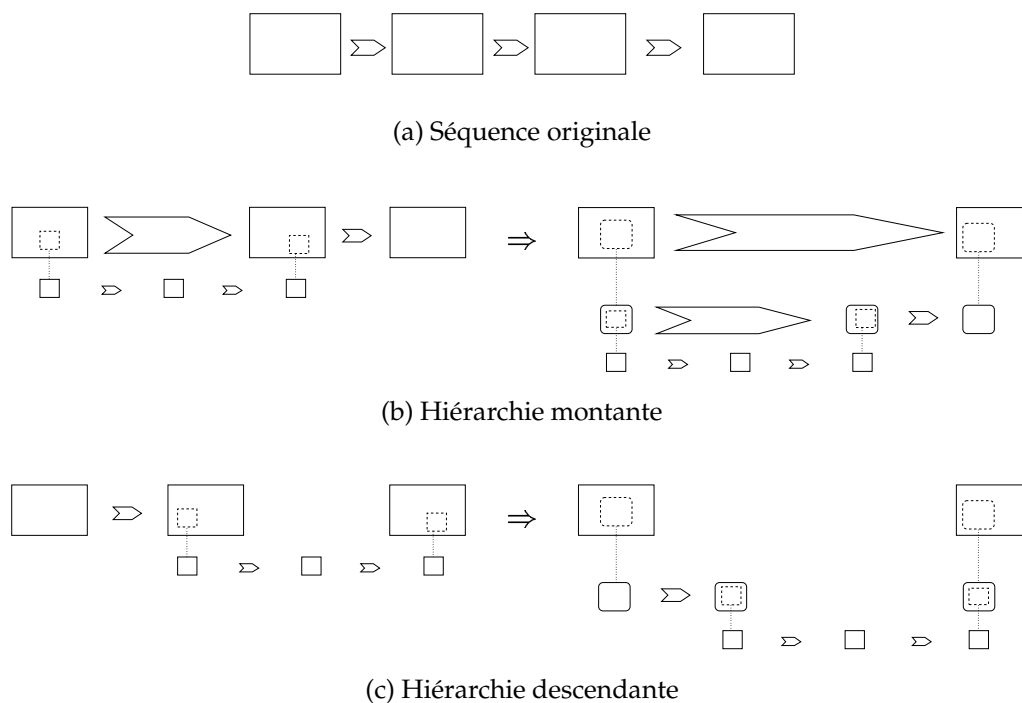


FIG. 3.11 – Exemple de différentes hiérarchies sur une même séquence de tâches

$$\begin{pmatrix} G^q \\ G^r \\ G^d \\ \mathbf{G} \end{pmatrix} = \begin{vmatrix} \Lambda & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \cdot \begin{pmatrix} G^q/\Lambda \\ \Lambda \\ G_r^d \\ \Lambda \\ G_o^d \\ \mathbf{G} \end{pmatrix} \cdot \begin{array}{cccccc} \Lambda & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

FIG. 3.12 – Le nouveau méta-ajustage est (G^q/Λ) . Nous avons directement reporté dans l'expression la distinction des méta-ajustages opérande/résultat en dupliquant le gabarit Λ .

Prenons comme exemple les deux tâches illustrées par la figure 3.10. La première tâche produit et consomme les points 1 par 1, alors que la seconde les utilisent 4 par 4 avec un glissement de 1 (la barre dans les gabarits divise les parties de pavage et d'ajustage) :

$$\begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1| \cdot \begin{pmatrix} 9 \\ \mathbf{G} \end{pmatrix} \cdot \overline{1} \quad \bowtie \quad \begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1 \ 1| \cdot \begin{pmatrix} 6 \\ \mathbf{G} \end{pmatrix} \cdot \overline{1 \ 0}$$

Après fusion, on a l'expression des dépendances QD ...

$$\begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1 \ 1| \cdot \begin{pmatrix} 6 \\ \mathbf{G} \end{pmatrix} \cdot \overline{1 \ 0}$$

... et la hiérarchie finale.

$$\begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1 \ 1| \cdot \begin{pmatrix} 6 \\ \mathbf{G} \end{pmatrix} \cdot \overline{1 \ 0} \left\{ \begin{array}{l} \begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1| \cdot \begin{pmatrix} 4 \\ \mathbf{G} \end{pmatrix} \cdot \overline{1} \\ \begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |1| \cdot \begin{pmatrix} 4 \\ \mathbf{G} \end{pmatrix} \cdot \overline{0} \end{array} \right.$$

On constate que les motifs de la deuxième tâche se recouvrent au trois-quarts. Ces motifs étant générés par la première tâche point par point, l'application hiérarchique recalcule 4 fois les mêmes valeurs. Si on divise par 3 le méta-pavage, on obtient une expression QD qui se simplifie¹⁰

$$\begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |3 \ 1 \ 1| \cdot \begin{pmatrix} 2 \\ 3 \\ 4 \\ \mathbf{G} \end{pmatrix} \cdot \overline{3 \ 1 \ 0} \xrightarrow{\text{simplification}} \begin{pmatrix} 9 \\ \mathbf{M} \end{pmatrix} \cdot |3 \ 1 \ 0| \cdot \begin{pmatrix} 2 \\ 6 \\ 3 \\ \mathbf{G} \end{pmatrix} \cdot \overline{3 \ 0 \ 1}$$

Ainsi, en exploitant les dépendances, les 3 macro-motifs de 12 points deviennent un macro-motif de 6 points et le taux de recalcul est divisé par 2.

► Aplatissage de hiérarchie

Si on agrandit une hiérarchie au maximum ($\Lambda = G^q$) alors la tâche supérieure n'a plus qu'une seule itération (un seul macro-motif). On peut alors la supprimer et faire

¹⁰En fait, seule la partie opérande se simplifie. On a donc, sur l'expression, découpé les deux ajustages

remonter sa hiérarchie d'un niveau. On appelle un tel procédé, un *aplatissement de hiérarchie*.

De tels aplatissements peuvent être intercalés entre les différentes opérations d'une fusion chaînée. Ce procédé permet, par exemple, de créer un seul niveau de hiérarchie sur toute une séquence de tâches.

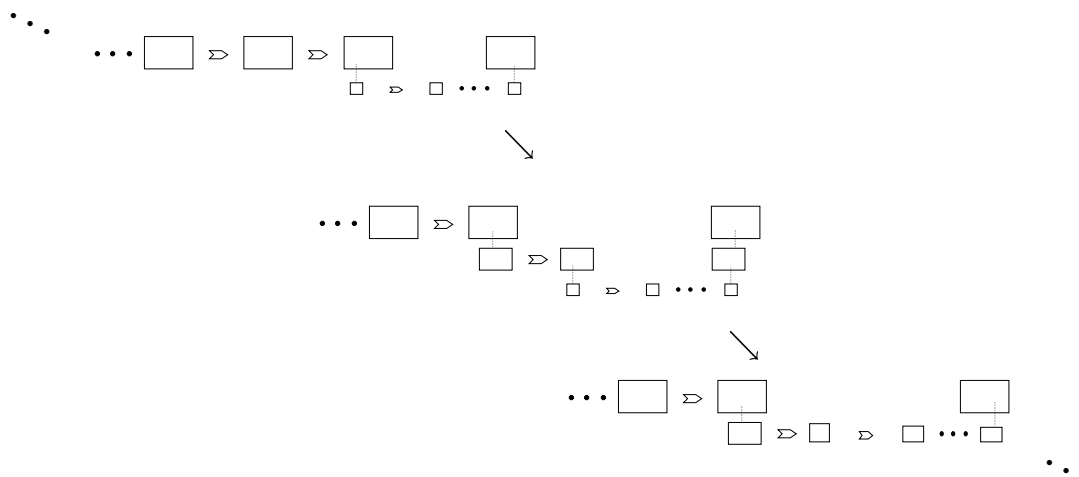


FIG. 3.13 – Hiérarchisation d'une séquence de tâches en un seul niveau

3.6 Stratégie de compilation

Par le biais des ODT, nous avons réalisé des outils élémentaires de transformations de codes ARRAY-OL vers du code ARRAY-OL hiérarchique : des tâches consécutives d'un même niveau d'application peuvent être reportées à un niveau inférieur, la taille des macro-motifs introduits restant variable.

Ces transformations n'induisent *a priori* aucun ordre d'application. Une multitude de schémas de transformations d'une application initiale sont donc envisageables. Pour transformer une application ARRAY-OL complète, il faut pouvoir répondre aux questions suivantes :

- Quelles sont les tâches qui descendent de niveau ?
- Quelles tâches appartiennent à une même sous-application ?
- Quel grain faut-il choisir pour les macro-motifs introduits ?

3.6.1 Dénombrement

Si on suppose qu'on a une chaîne de n tâches linéaires, on peut former $(n-1)$ couples de tâches pour appliquer une fusion. Ceci nous donnera, en résultat, une chaîne de

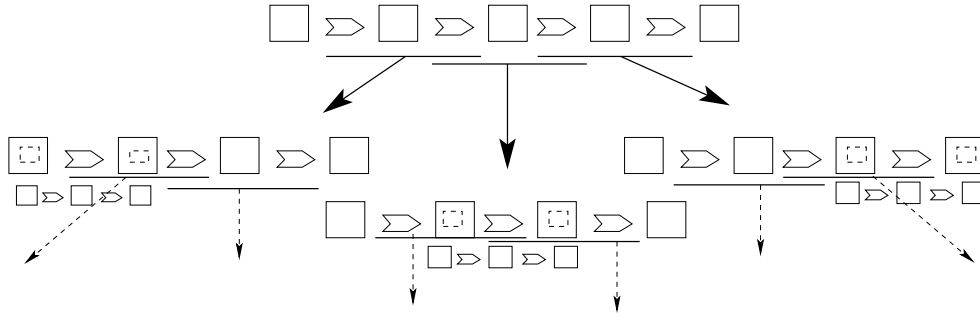


FIG. 3.14 – Exemple de solutions de fusion

$(n - 1)$ tâches. On peut s'arrêter là mais si on décide de continuer, on a alors $(n - 2)$ couples possibles. Si on continue le raisonnement, on obtient finalement

$$\sum_{i=1}^{n-1} \frac{(n-1)!}{(n-i)!} < e \cdot (n-1)!$$

soit $\mathcal{O}((n-1)!)$ schémas possibles¹¹. Sachant que, de plus, on peut aplatir toute hiérarchie qui a été formée par une fusion, la combinaison de ces deux transformations nous donne

$$\sum_{i=1}^{n-1} \frac{2^i (n-1)!}{(n-i)!} = 2^n (n-1)! \times \sum_{i=1}^{n-1} \frac{(1/2)^i}{i!} < 2^n \cdot \sqrt{e} \cdot (n-1)!$$

soit $\mathcal{O}(2^n (n-1)!)$ schémas possibles.

L'explosion combinatoire à laquelle on arrive, nous oblige effectivement mettre en place une stratégie d'utilisation de ces outils pour approcher (converger vers) la solution la meilleure en fonction de critères eux-mêmes à définir.

Plusieurs stratégies ont été testées sur la VBL. De cette expérience, nous retiendrons quelques réflexions sur la spécification des macro-motifs.

3.6.2 Hiérarchisation de la Veille Large Bande

Les exemples de hiérarchisations de la VBL présentés maintenant sont illustrés par l'interface graphique GASPARD présentée au chapitre 4.

À partir de la spécification de la VBL et en utilisant manuellement les transformations construites sur les ODT, nous avons proposé une version hiérarchisée. L'annexe du rapport des travaux LIFL/TMS [DDMS99] reprend les codes produits pour ces trois réécritures de VBL. L'annexe du présent manuscrit donne le code initial (A.1) et le code final (A.2).

¹¹Ce comptage n'évite pas les redondances : on compte plusieurs fois les fusions sur des ensembles disjoints de tâches. Il s'agit juste d'un ordre de grandeur.

► Taille mémoire nécessaire pour une application hiérarchique

Dans les exemples de hiérarchisations que nous allons présenter, il sera question de la taille requise pour l'exécution de telle ou telle application. Or la taille mémoire nécessaire à l'exécution d'une application ARRAY-OL hiérarchique est fonction de la structure du code généré par le compilateur.

Dans le schéma de compilation actuel, la place nécessaire équivaut à la somme de tous les tableaux (y compris ceux des niveaux hiérarchiques inférieurs). La mise en œuvre du court-circuit (cf. la section 2.3.2) permet de réduire cette taille mémoire à la somme des tableaux apparaissant à leur plus haut niveau de hiérarchie. De plus, les tableaux d'entrée/sortie nous sont imposés et, en simulation, on peut les laisser sur un support externe. On ne prendra donc pas leur taille en compte dans la suite.

► Traitement de l'infini

Dans un premier temps nous avons appliqué une hiérarchisation complète de l'application VBL¹², opération rendue nécessaire par la présence d'un flux infini (dimension temporelle). Les deux niveaux supérieurs de l'application produite sont présentés à la figure 3.15.

Le tableau final est construit point par point en appliquant pour chacun la chaîne complète de traitement. La taille total des tableaux à stocker en mémoire représente 60 Mo.

Cette valeur peut paraître acceptable pour une station de travail récente en mono-utilisation. En fait, la tâche de `FORMATION_DE_VOIE` réutilise un même point opérande plusieurs fois. Les motifs nécessaires pour le calcul de deux points voisins du tableau final se recouvrent de manière dramatique¹³ et entraîne le recalcul de nombreux points (cf. 3.5.2). Le gain mémoire coûte ici très cher en temps de calcul.

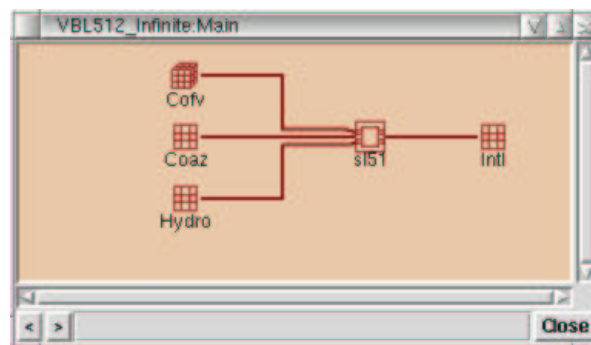
► Réduction des recouvrements

Vu le taux de recouvrement et l'utilisation cyclique de la dimension des hydrophones, il a semblé nécessaire de faire descendre toute cette dimension dans la hiérarchie avec la technique d'agrandissement du macro-motif présentée en 3.5.3.

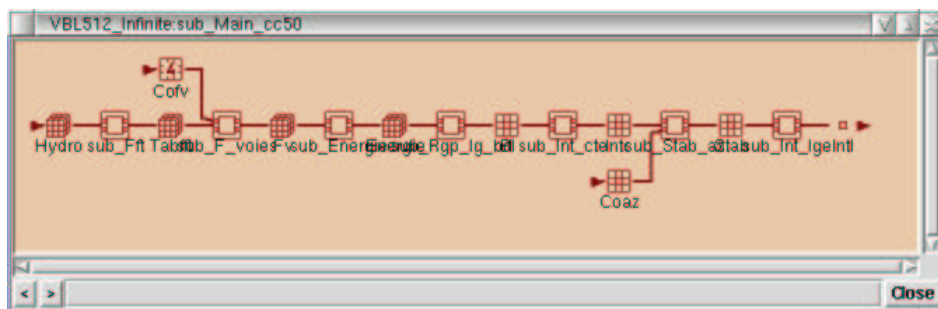
La taille du macro-motif d'entrée est multipliée par trois (on passe de 192 à 512 hydrophones) et le reste de la sous-application associée produit 128 voies au lieu des 8 précédemment. Cela nous donne une taille mémoire requise de 170 Mo mais les recalculs sont supprimés.

¹²Il s'agit de l'enchaînement fusion/aplatissement vu en 3.5.3.

¹³On décale de 4 le motif représentant 200 hydrophones à chaque itération. On obtient un taux de recouvrement de 98%.

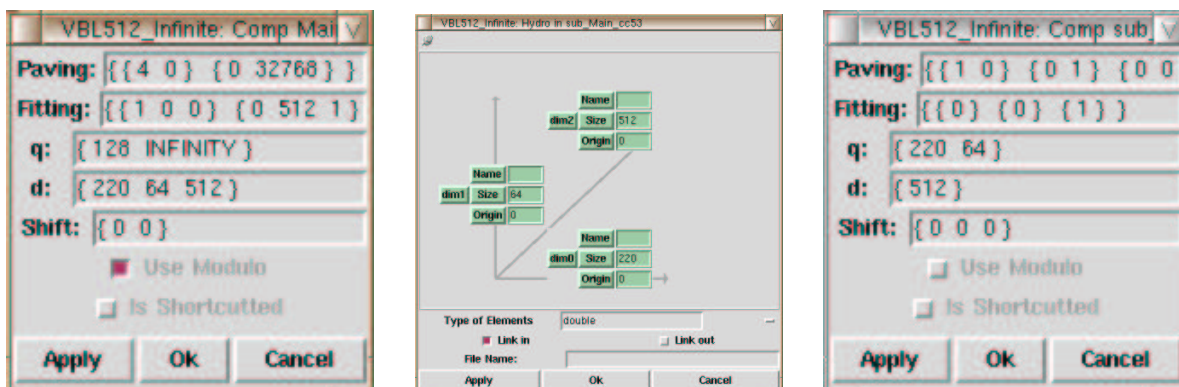


(a) Application supérieure

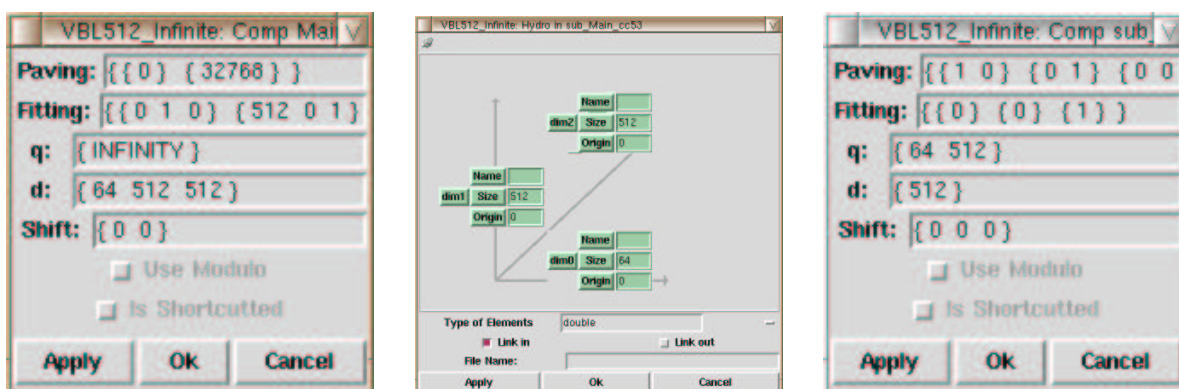


(b) Application inférieure

FIG. 3.15 – Application résultat d'une hiérarchisation complète de la VBL



(a) Tableau et pavage/ajustage initiaux



(b) Tableau et pavage/ajustage après agrandissement du macro-motif

FIG. 3.16 – Agrandissement du macro-motif à toute la dimension

Remarque Cette transformation illustre une caractéristique appelée « *corner-turn* ». Les motifs opérande et résultat sur le tableau intermédiaire, TABFFT, sont orthogonaux. Un motif opérande nécessite un grand nombre de motifs résultats qui eux-mêmes permettraient de calculer plusieurs motifs opérandes (figure 3.17).

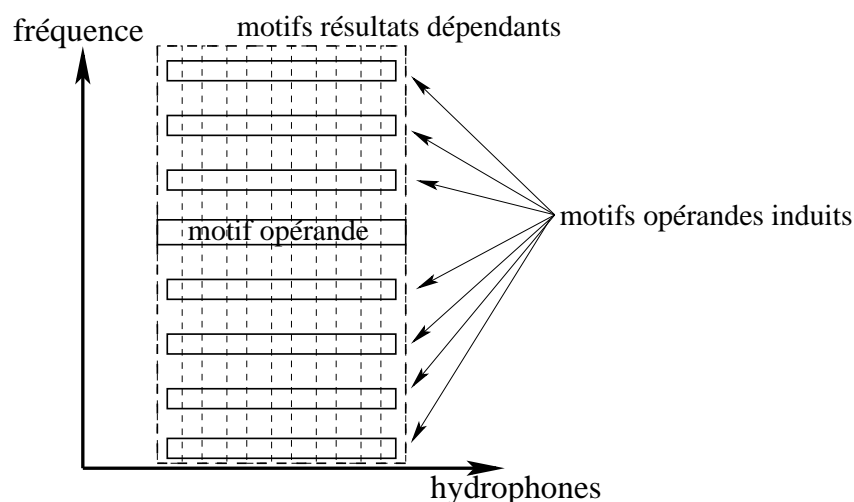


FIG. 3.17 – Illustration du phénomène de corner-turn

Pour des raisons liées au modèle ARRAY-OL, on ne peut pas calculer un seul motif opérande à la fois car le décalage entre l'origine des motifs opérandes et résultats n'est pas constante (cf 3.4.7). La sous-application doit donc inclure tous ces motifs dans ses calculs, ce qui entraîne une importante augmentation des macro-motifs en aval de la hiérarchisation.

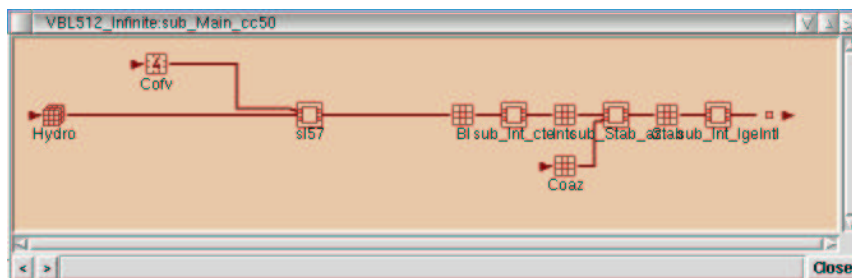
► Réduction des ressources temporaires

Généralement une application de traitement de signal a pour rôle d'extraire d'un grand nombre de données en entrée, des informations pertinentes susceptibles d'être utilisées dans le traitement de données qui s'en suivra. Cela signifie que normalement les tableaux d'entrée sont bien plus grands que ceux de sortie. C'est notamment le cas de la VBL puisque les deux premiers tableaux (sur un nombre fixe de récurrences) sont à peu près 128.000 fois plus grands que le tableau final.

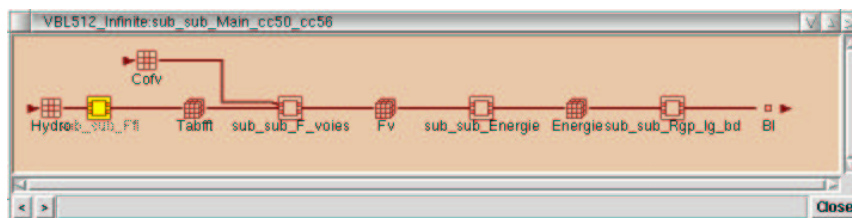
Pour ce type d'application, à l'évidence la stratégie la plus efficace consiste à hiérarchiser au plus profond les tableaux de début de chaîne et de laisser entiers ceux de la fin.

Nous avons réalisé une nouvelle hiérarchisation jusqu'au tableau BL. C'est à partir de ce tableau que les tailles décroissent de plusieurs grandeurs : tous les tableaux en amont sont de tailles supérieures à la dizaine de Mo alors que la somme de ceux en aval est inférieure à 100 Ko. La place mémoire nécessaire du troisième niveau hiérarchique

est alors de 800 Ko mais les recalculs induits par le recouvrement de motifs demeurent (la tâche de `FORMATION_DE_VOIE` fait encore partie de la branche hiérarchisée). Le résultat de cette nouvelle transformation est illustré à la figure 3.18.



(a) Première hiérarchie



(b) Deuxième hiérarchie

FIG. 3.18 – Nouvelle hiérarchisation de la VBL

Comme plus haut, nous descendons la dimension des hydrophones, ce qui nous donne finalement une taille mémoire de 3 Mo et 2 niveaux de hiérarchie.

3.6.3 Critères objectifs

Les deux principaux critères de transformation sont la réduction de l'espace mémoire nécessaire et des calculs redondants, auxquels on peut ajouter la réduction des communications en mémoires distribuées (fortement liée d'ailleurs à la réduction de l'espace). Les deux premiers critères cités étant plutôt antagonistes, le résultat optimal est un compromis entre ces deux là. Pour la plupart des applications le temps d'exécution est souvent considéré comme la caractéristique principale à minimiser cependant, en traitement du signal, il arrive fréquemment que le support d'exécution doivent subir des contraintes beaucoup plus grandes qu'une simple station de travail (cas des machines embarquées). Dans ce cas, les contraintes physiques comme la place mémoire, les temps de latence et le débit des données prennent alors le pas sur le temps de calcul.

Les transformations constituent notre moyen d'optimisation des applications

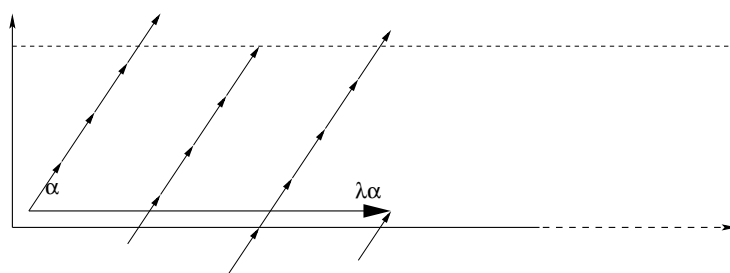


FIG. 3.19 – Isolation du vecteur infini

ARRAY-OL. Malheureusement le champs d'exploration est combinatoire et nous ne pouvons pas prédire le résultat d'une série de transformations sans la réaliser effectivement. La stratégie semble ne pouvoir éviter une recherche quasi-exhaustive (genre *branch & bound*). Cependant quelques considérations peuvent nous permettre, sinon de nous guider, au moins de restreindre l'espace de recherche. Il s'agit essentiellement de borner les critères de mémoires et de calculs dans une branche hiérarchique.

► Ratio des bornes QD infinies

Dans le cas où les bornes d'indices de pavage et d'ajustage sont finies, il est facile de déduire le nombre de TE à calculer et de là le ratio entre celui des tâches d'origines avec celui des tâches transformées. Quand une de ces bornes est infinie, la notion même de ce ratio devient moins claire. Cependant on peut encore en trouver une interprétation assez réaliste en le reliant à la quantité de points nécessaires aux calculs des TE : à tranche d'espace fixée tendant vers l'infini combien de TE peut-on calculer ?

La raison d'une telle approche se justifie par le fait que l'infini décrie souvent l'existence d'un flux de donnée. Le ratio en question s'apparente alors au nombre moyen de TE en *régime continu*. Pour cela il faut donc synchroniser les bornes infinies des tâches sur une même valeur spatiale.

Soit une séquence de deux tâches infinies, T_1 et T_2 . Avant d'appliquer la fusion, on va synchroniser les deux bornes d'itérations infinies. On commence par isoler pour chaque tâche la dimension spatiale infinie du vecteur de pavage sur le tableau intermédiaire (i.e. la partie résultat de la première tâche et la partie opérande de la seconde) en l'itérant un certain nombre de fois de façon à annuler les autres dimensions grâce au modulo (cf. la figure 3.19) :

$$\begin{aligned}
\begin{pmatrix} \infty \\ M \\ \mathbf{M} \end{pmatrix} \cdot \left| \begin{array}{cc} \alpha & L \\ C & \mathcal{P} \end{array} \right| \cdot \begin{pmatrix} \infty \\ Q \\ \mathbf{G} \end{pmatrix} &= \begin{pmatrix} \infty \\ M \\ \mathbf{M} \end{pmatrix} \cdot \left| \begin{array}{cc} \alpha & L \\ C & \mathcal{P} \end{array} \right| \cdot \left| \begin{array}{ccc} \lambda & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array} \right| \cdot \begin{pmatrix} \infty \\ \lambda \\ Q \\ \mathbf{G} \end{pmatrix} \cdot \overline{\begin{array}{ccc} \lambda & 1 & \mathbf{0} \\ 0 & 0 & \mathbf{1} \end{array}} \\
&= \begin{pmatrix} \infty \\ M \\ \mathbf{M} \end{pmatrix} \cdot \left| \begin{array}{ccc} \lambda\alpha & \alpha & L \\ \mathbf{0} & C & P \end{array} \right| \cdot \begin{pmatrix} \infty \\ \lambda \\ Q \\ \mathbf{G} \end{pmatrix} \cdot \overline{\begin{array}{ccc} \lambda & 1 & \mathbf{0} \\ 0 & 0 & \mathbf{1} \end{array}}.
\end{aligned}$$

avec $\lambda = \bigvee_i \frac{(C_i \vee M_i)}{C_i}$ de telle sorte que $(\lambda C =_{(\text{mod } M)} 0)$.

Ensuite on segmente à nouveau la dimension d'itération infinie des deux tâches de façon que l'incrément des vecteurs de pavage infinis sur le tableau intermédiaire soit le même (ce sera le PPCM des deux précédents).

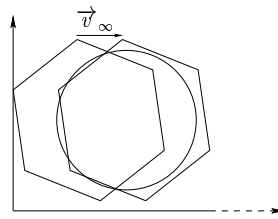
Remarque : On suppose que l'infinité du tableau est utilisé i.e. $\alpha \neq 0$. Si ce n'est pas le cas alors, après la synchronisation, on peut éliminer purement et simplement le pavage infini des deux côtés et borner la dimension infinie du tableau. En effet, si le résultat est nul alors l'opérande aussi sinon on utiliserait des points qui ne sont pas produits, et si l'opérande est nulle alors on utilise une partie finie du tableau et on peut donc remplacer l'itération infinie du résultat par une borne finie.

On a finalement deux tâches dont l'itération infinie consiste en un même déplacement sur la dimension infinie. Considérons un macro-motif constitué par l'union des motifs correspondants aux itérations de pavage finies. Pour effectuer un macro-motif de la seconde tâche, il faut un ou plusieurs macro-motifs de la première (disons les itérations de 0 à n). Pour effectuer les N premiers macro-motifs de la seconde tâche, il faudra alors les $N + n$ ($= \bigcup_{0 \leq i \leq N} [i \dots i + n]$) macro-motifs de la première (figure 3.20). Si N est assez grand devant n , le rapport des tailles requises tend vers 1.

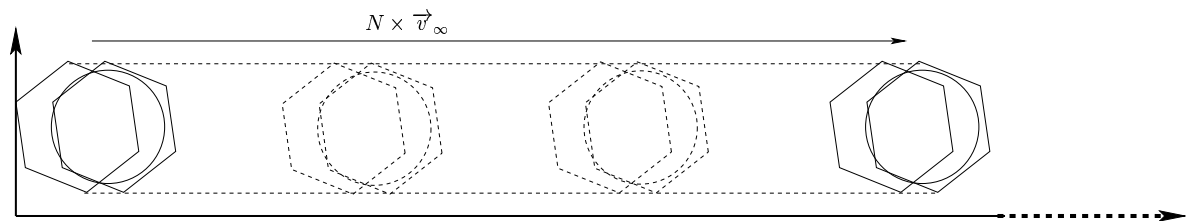
Après une fusion de ces deux tâches, la borne d'itération infinie va se retrouver forcément dans le macro-pavage. Les valeurs des deux vecteurs de pavage (opérande et résultat) associés à l'itérateur infini dans l'expression des dépendances QD ne peuvent qu'être égales. Sinon il se créerait un décalage entre l'opérande et le résultat qui tendrait vers l'infini à cause de l'absence de borne. Cela est impossible puisque le macro-motif résultat doit être inclus dans celui de l'opérande. En segmentant une nouvelle fois les vecteurs de pavage infinis, on peut annuler toutes les composantes non infinies des vecteurs, ce qui nous donne :

$$\cdots \left| \begin{array}{cc} \beta & L'_o \\ \mathbf{0} & \mathcal{P}'_o \end{array} \right| \cdot \begin{pmatrix} \infty \\ Q' \\ D' \\ \mathbf{G} \end{pmatrix} \cdot \overline{\begin{array}{cc} \beta & L'_r \\ \mathbf{0} & \mathcal{P}'_r \end{array}} \cdots$$

L'itérateur infini est donc, dans un certain sens, équivalent entre l'opérande et le



(a) Les différences entre macro-motifs opérandes/résultats...



(b) ... s'estompent en régime continu.

FIG. 3.20 – Régime continu sur l'infini

résultat et la valeur commune, β , sur le vecteur de macro-pavage (au niveau des dépendances QD , avant le retour au spatial) précise le ratio entre cet itérateur et l'itérateur infini des tâches originelles. Ceci permet de calculer le taux de redondance après fusion grâce à la formule classique du nombre de TE en remplaçant le rapport des infinis par cette valeur.

Si l'itérateur de pavage d'une sous-tâche est $\begin{pmatrix} q \\ D \\ G \end{pmatrix}$ alors son ratio sera $Q\beta/Q'q$.

► Ratio des tailles de tableau infini

La techniques décrites au paragraphe précédent permet d'évaluer le taux de redondance de calculs lors d'une fusion. Pour ce qui est du recouvrement spatial sur les macro-motifs, on peut raisonnablement l'évaluer comme le rapport entre la taille du tableau d'origine et la taille du tableau réduit (qui n'est pas forcément la taille du macro-motif) multipliée par le nombre de macro-motifs i.e. le nombre d'itérations de la tâche supérieure.

Si on traite des tableaux infinis, on peut utiliser la technique précédente qui nous donne au final, grâce à un double rapport (celui de l'itérateur infini du macro-pavage vers l'itérateur infini du pavage original et celui de l'itérateur infini du pavage original vers la dimension infinie du tableau), le ratio entre l'itérateur infini du macro-pavage et

la dimension infinie du tableau. Avec les notations précédentes, ce ratio serait $\alpha\beta$.

► Taux de recouvrement linéaire

Nous venons de voir comment on peut évaluer le résultat d'une hiérarchisation en terme de réduction d'espace et de recouvrements d'espace et de calculs. Mais il serait bon aussi de savoir comment ces critères évoluent quand on utilise la transformation d'agrandissement du macro-motif.

L'évolution étant fonction des matrices de changement d'espaces QD , il n'est pas aisé de décrire le résultat sans avoir effectué réellement la transformation. Cependant on peut avoir des indications sur l'influence de certains vecteurs dans des cas précis. Notamment, on peut détecter les vecteurs qui provoquent des recouvrements linéaires ainsi que des recouvrements cycliques (présence de vecteurs colinéaires dans les parties d'ajustage et de pavage).

► Borne des critères dans le dénombrement

La fusion de deux tâches consécutives permet d'atteindre la réduction maximale sur le tableau intermédiaire¹⁴. En effet les fusions suivantes s'appliqueront éventuellement à la tâche supérieure mais les deux tâches inférieures ainsi que le tableau intermédiaire demeureront inchangés. Donc la somme des réductions consécutives aux fusions des tâches deux par deux donne une borne inférieure au gain espéré. Cette borne pouvant bien sûr ne pas être atteinte. Une fois qu'on a décidé d'une transformation, on peut ré-évaluer la borne puisque les deux tâches impliquées dans la fusion ne peuvent plus fusionner indépendamment l'une de l'autre. De la même façon, une borne maximale des calculs redondants peut être trouvée.

3.6.4 Expérience de compilation parallèle

Le schéma de description d'ARRAY-OL exhibe déjà du parallélisme par définition. Il y a celui des tâches dans le modèle global et plus particulièrement le parallélisme massif de données dans la construction des tâches. Il semble alors facile d'exploiter ces caractéristiques pour générer du code parallèle. Les questions qui demeurent sont liées au fait de savoir comment les exploiter au mieux c'est-à-dire jusqu'à quel point et où paralléliser les calculs et, comment distribuer les calculs et les données. Évidemment, les critères déjà définis s'appliquent également au code parallèle (plus les ressources temporaires et les redondances sont faibles, plus le code est efficace), mais nous nous intéressons ici aux facteurs spécifiques des machines parallèles i.e. l'exploitation des

¹⁴maximale dans le cadre des algorithmes et des outils à disposition

ressources parallèles et les problèmes liés aux architectures *NUMA* (communications, synchronisation, recouvrement de tâches...).

► Parallélisme sur mémoire partagée

En SMP, il n'y a pas beaucoup de changement, la réduction de l'occupation mémoire et des redondances de calculs demeurent le cœur du problème. Les tâches *ARRAY-OL* étant par nature massivement data-parallèle, un code SMP peut se contenter de découper l'espace *QD* en parts égales et d'attribuer un thread par processeur. Pour des stations de travail classiques, ce nombre dépassant rarement la dizaine, il n'y a pas vraiment de problème de famine.

Les seules réserves à apporter concernent les tâches qui comporteraient des traitements irréguliers (collection, entrée/sortie ...). Dans ce cas la re-synchronisation des threads en fin de tâche pourrait être pénalisante. On peut alors se demander si une hiérarchisation pourrait s'avérer bénéfique en permettant de lancer les threads sur la tâche supérieure. Ceux-ci s'exécuteraient sans synchronisation sur toutes la sous-hiérarchie. L'inconvénient cependant vient de l'obligation de dupliquer les ressources de la sous-hiérarchie pour chaque thread.

Une génération de code multithreadé a été intégrée à la plate-forme *GASPARD*, notamment par T. *POUPART* durant son stage de DEA [Pou99]. Le schéma de génération est très classique (division de l'espace *QD* en parts égales), là encore la stratégie repose sur la transformation de code pour contrôler l'emploi du parallélisme.

Des série de tests ont été réalisés sur le passage en multithreadé de la *VBL* sur un quadri-processeurs *Xéons* (cf. figure 3.21). L'application testée correspond à la hiérarchie des trois premières tâches de la *VBL* (cf. le code en annexe A.3). Les deux critères étaient le comportement de l'application suivant le nombre de threads et suivant le placement de ces threads (au niveau des tâches opératives ou au niveau de l'application supérieure).

On remarque d'abord que le placement des threads influe sur les performances comme on l'avait prévu ($\approx 12\%$ à partir de 5 threads). L'écart peut sembler peu important alors que les threads de chaque sous-tâches sont arrêtés et relancés 128 fois (nombre de pavage de la tâche supérieure). Cela s'explique par le faible coût de gestion d'un thread (à peu près $30\mu s$ pour la création et $50\mu s$ pour la création et l'arrêt sans attente) et la faible désynchronisation due à la régularité de l'application.

Par contre on a un très bon rendement puisqu'on arrive à une accélération supérieure à 3, 4 entre 4 threads (qui est le nombre de processeurs) et 16 threads avec le maximum à 3, 8 pour 5 threads ce qui doit correspondre à un compris optimal entre le recouvrement des threads et le surcoût de gestion de la bibliothèque.

Pour évaluer l'impact des threads en terme de recouvrements de processus, nous avons également testé cette version sur une machine mono-processeur (*PIII/850MHz*).

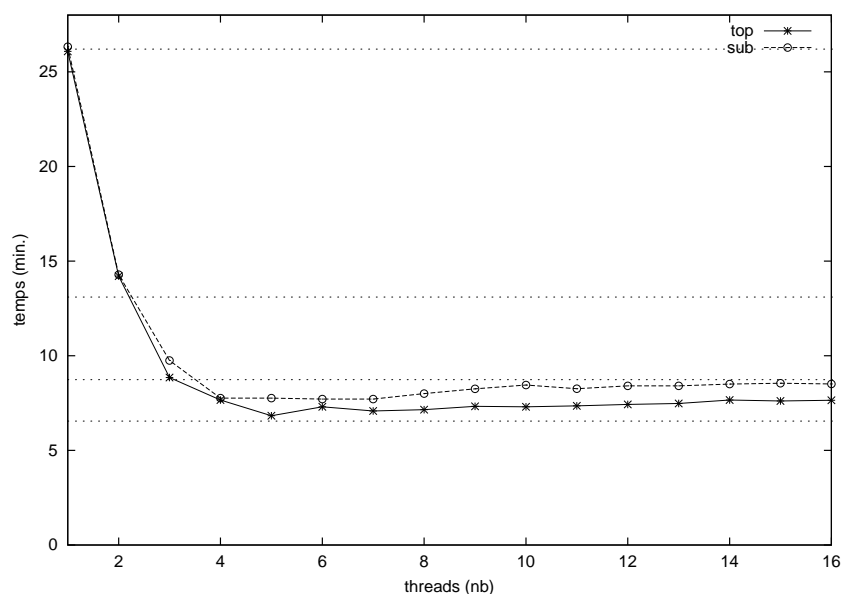


FIG. 3.21 – Performances d'un extrait de la VBL hiérarchique multithreadée sur un quadri-xéons. Pour la courbe *top*, les threads sont placés dans la tâche supérieure. Pour la courbe *sub*, les threads sont placés dans les trois sous-tâches. Les traits horizontaux correspondent au temps séquentiel, à la moitié, au tiers et au quart.

Il y a effectivement une légère amélioration mais elle ne dépasse pas 2% du temps initial.

► Parallélisme sur mémoires distribuées

Sur mémoires distribuées, le point crucial concerne le nombre de communications qui est fortement lié à la répartition des données. Il vaut mieux donc éviter d'exploiter le parallélisme au niveau des motifs comme c'était le cas en SMP.

Une solution consiste simplement à distribuer les tâches complètes sur les unités de calculs. Cela signifie aussi de rapatrier les tableaux opérands avant et de retourner les tableaux résultats. Dans ce cas, il serait préférable que les unités effectuent des séquences de tâches complètes pour permettre une réutilisation locale des tableaux générés. Une telle décomposition n'est pas toujours aisée à trouver. La VBL, par exemple, est constituée d'une seule séquence de tâche et n'autorise pas un tel procédé.

La hiérarchisation peut là encore s'appliquer puisque un niveau de hiérarchie (i.e. une sous-application) constitue un ensemble de données auto-suffisantes. Il faut alors contrôler, de la même façon que pour les calculs, le recouvrement sur les données en jouant sur l'agrandissement des macro-motifs.

Un module de génération de code distribué a été réalisé dans l'environnement GASPARD. Il utilise la bibliothèque de passage de messages PVM. Le processus maître

(i.e. le programme de lancement) créé autant de processus fils que l'utilisateur a demandé en argument. Le processus maître détient les tableaux initiaux et effectue les tâches non distribuées. Pour une tâche distribuée, il est chargé d'envoyer aux fils les motifs à traiter et de recevoir les résultats, en plus d'effectuer sa propre partie de la tâche. Pour ne pas pénaliser la tâche du processus maître, un thread est créé pour assurer l'envoi et la réception des motifs.

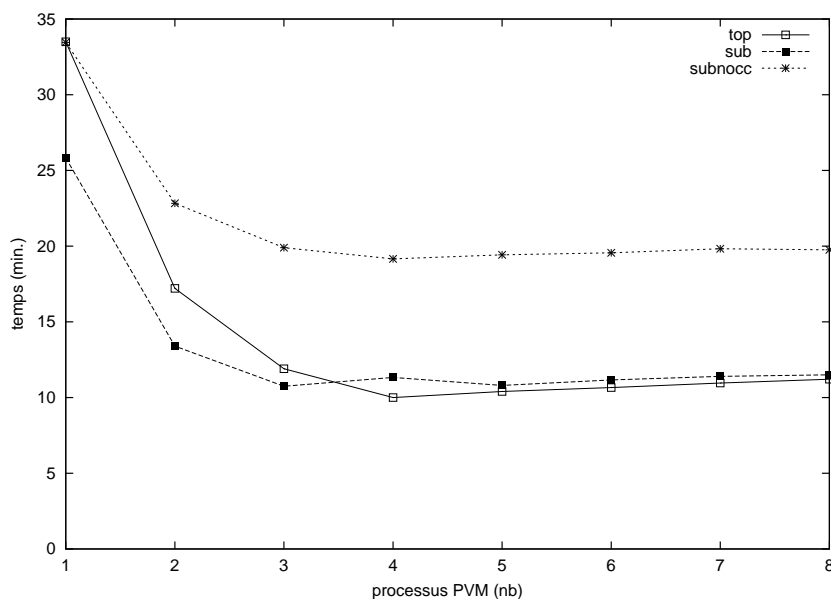


FIG. 3.22 – Performances d'un extrait de la VBL hiérarchique distribuée sur un quadri-xéons. Pour la courbe top , seule la tâche supérieure est distribuée (cela a pour conséquence d'annuler le court-circuit du passage des tableaux dans la hiérarchie). Pour la courbe sub , seules les trois sous-tâches sont distribuées. Pour la courbe $subnocc$, la distribution est la même que sub mais on a éliminé le court-circuit sur les tableaux supérieurs.

Des séries de tests ont été effectuées sur le même extrait de VBL que plus haut (cf. figure 3.22). Ils ont été réalisés sur le Quadri-xéons en utilisant des processus PVM communicants par la couche réseau. Il s'agit là encore de mesurer l'impact du placement de la distribution et du nombre d'unités de calculs. Les tests ont été menés jusqu'à 8 processus bien qu'il n'est pas réaliste d'avoir plus de processus que de ressources (4 dans notre cas).

On constate tout d'abord que l'application sub part avec un avantage puisqu'elle n'a pas à allouer les tableaux d'entrée/sortie de la sous-hiérarchie grâce au court-circuit. Cependant elle perd rapidement cette avance alors que la distribution intervient, ce qui confirme nos prévisions. À partir de 4 processus, d'ailleurs, l'application top dépasse l'application sub . L'écart est plus flagrant avec l'application $subnocc$ alors que les deux applications requièrent les mêmes ressources mémoires. L'effet se joue sur le nombre de

communications : 112 Mo en 256 motifs pour `top` contre 1,4 Go en 880000 motifs pour les autres applications.

3.7 Conclusion

Le langage ARRAY-OL est bien adapté pour spécifier les algorithmes des applications TS mais ne propose pas de directives de compilation hormis celle, simpliste, présentée précédemment. En prenant compte le fait que cette version est théoriquement la plus efficace sur machine à mémoire partagée infinie et que le langage est destiné à devenir un modèle de référence pour le domaine TS, il nous a semblé judicieux de proposer une stratégie de compilation basée sur la transformation du code ARRAY-OL en vue de son exécution sur un support optimisé pour ce langage (station de travail ou machine dédiée).

La hiérarchie est alors un bon moyen d'introduire des ordonnancements différents faisant intervenir toutes les tâches d'une application. Pour formaliser ces transformations, il nous fallait un modèle qui puisse rendre compte des dépendances de données tout en restant proche d'ARRAY-OL. Ce modèle des ODT est constitué de combinaisons d'opérateurs élémentaires de relations entre espaces amenant à une expression simple d'une tâche ou d'une séquence de tâches ARRAY-OL.

Cela nous a permis de décrire la construction formelle d'une hiérarchie à partir d'une séquence de deux tâches, la fusion, ainsi que, par extension, deux autres opérations pour contrôler la taille des hiérarchies créées. La possibilité d'enchaîner les fusions en différents endroits d'une application, en y intercalant les autres opérateurs, nous a amené à nous interroger sur les moyens de diriger l'ordre d'application des transformations. Dans ce sens, nous avons défini certains critères mesurables rendant compte des résultats ou de l'influence de ces transformations.

En séquentielle, les effets de la réduction des ressources mémoires ou des redondances de calculs paraissent évident sur l'efficacité du code. Sur machine parallèle, d'autres facteurs rentrent en ligne de compte (degré de parallélisme, distribution...). Nous avons donc mené spécifiquement des tests en mémoire partagée et en mémoires distribuées qui se sont révélés assez positifs.

Notre souhait étant de rendre accessible ces transformations aux concepteurs d'applications TS, l'idée de les intégrer dans un atelier visuel de développement nous est apparue nécessaire. Cela a abouti à l'environnement GASPARD décrit au chapitre suivant.

Chapitre 4

Transformations visuelles dans l'environnement GASPARD

4.1 Introduction

La partie transformationnelle d'une application de traitement de signal ne constitue qu'une étape de la chaîne de traitement complète. Historiquement, la saisie du modèle global d'ARRAY-OL se faisait à l'aide d'un domaine de PTOLEMY (cf. chapitre 2). Ensuite TMS utilisait un environnement écrit en ITCL (extension objet du langage TCL) pour la spécification du modèle local, pavage et ajustage. En sortie, PTOLEMY pouvait produire un fichier au format ARRAY-OL textuel.

Hormis le fait que l'environnement était peu agréable à utiliser, la séparation du modèle global intégré à PTOLEMY et du modèle local qui opérait comme une boîte noire ne pouvait favoriser la transformation interactive d'applications que requière un atelier complet de développement et ne permettait pas de maîtriser la processus complet de compilation.

Dans le cadre de la collaboration entre le LIFL et TMS, le travail de l'équipe a également consisté à lancer la création d'un atelier graphique de développement et de simulation d'application TS en ARRAY-OL. GASPARD hérite en grande partie de la première version d'atelier de programmation visuelle. Il conserve les caractéristiques principales en y ajoutant une convivialité d'utilisation basée sur la réutilisabilité des composants logiciels. Il est complètement écrit avec des outils du domaine public, libérant l'utilisateur des contraintes d'utilisation liées à PTOLEMY.

Bien qu'il soit en premier lieu spécifique à ce langage, l'interface GASPARD doit à terme supporter d'autres types d'applications, en particulier celles intervenant dans une chaîne sonar i.e. le traitement de données intensif sur des collections (ce travail est en cours de réalisation avec la version 2 de GASPARD). Des bibliothèques de composants spécialisés pour tel ou tel domaine d'applications sont également prédéfinies.

GASPARD définit un exécuteur permettant une exécution sur station de travail et aussi sur machine multithreadée. Le code produit utilise dans ce cas la bibliothèque Pthread. Ces phases de compilation sont rendues possibles par la manipulation interactive des codes sources via les outils de transformation proposés au chapitre 3.

La version présentée ici reste préliminaire, elle permet d'éditer et de visualiser des applications ARRAY-OL ainsi que d'effectuer les transformations précédemment décrites et de générer du code. Une version en cours de développement devrait étendre les fonctionnalités de l'interface [BDDM01].

Dans ce chapitre, nous présentons les concepts visuels introduits comme définition de la métaphore de GASPARD. Puis nous présenterons les outils utilisés pour réaliser cette interface. Enfin un exemple basé sur la VBL permettra de valider visuellement les transformations de code.

4.2 L'environnement GASPARD

L'environnement de spécification visuel GASPARD est construit à partir des caractéristiques héritées du modèle ARRAY-OL. Le modèle global est représenté par une description visuelle d'un composant logiciel. La métaphore que nous utilisons repose sur le « design » d'un composant électronique de type circuit imprimé. Les tableaux sont des informations circulant sur des fils, ceux-ci sont connectés à différents slots sur lesquels sont branchés d'autres composants logiciels. Cette phase de branchement permet de spécifier les informations d'ajustage et de pavage qui indiquent comment le composant ainsi connecté référencera les tableaux liés au slot. Les motifs obtenus par ce pavage et ajustage deviennent les tableaux en entrée/sortie du composant ainsi branché.

De cette façon chaque composant peut être spécifié indépendamment des composants qu'il utilise et vice versa. En conséquence, les composants sont tous réutilisables. Les composants de GASPARD sont assimilés à des classes telles que définies dans des langages à objets. Seules les instances de ces classes seront dites exécutables. Des bibliothèques de composants prédéfinis, éventuellement génériques, peuvent être créées pour des types d'applications particulières.

Les dépendances entre différents composants connectés dans le même composant logiciel (niveau global) sont exprimées par les fils entre les slots. Ceux-ci véhiculent les tableaux en entrée et en sortie. Le compilateur a en charge d'y extraire un schéma d'exécution en respectant la sémantique.

Trois aspects méritent d'être mieux précisés : le composant, le liens, et la notion de complétude.

Composant Un *composant* correspond à une action encapsulée sur la totalité ou une partie des tableaux. Nous identifions deux types de composants :

- Un *composant primaire* n'est pas défini à l'aide d'autres composants. Il est décrit par une fonction C++ (éventuellement multithreadée). Cette fonction accepte des tableaux en entrée, et produit des tableaux en sortie. Ses paramètres d'entrée/sortie sont les liens des composants. La fonction doit permettre une exécution SPMD sans conflit.
- Un *composant composé* utilise plusieurs composants primaires ou composés connectés sur des slots eux-mêmes reliés par des fils de communications

Lien Certains tableaux d'un composant sont particuliers : ils caractérisent les liens d'entrée/sortie :

- un *lien d'entrée* est un tableau qui est ni produit par le composant, ni associé à un fichier, ni initialisé comme un tableau constant ;
- un *lien de sortie* est un tableau qui est ni consommé dans le composant, ni associé à un fichier.

Ces liens définissent l'interface du composant ; ils seront par la suite associés aux tableaux du composant qui recevra ce dernier lors du branchement dans un slot. Les composants sont réguliers : la taille et la forme des liens sont statiquement définies, une approche dynamique a été introduite dans [BDD⁺99]. Les liens et le composant sont considérés comme soudés et sont donc indissociables (notion de « pins » et de « chip »).

Complétude d'un composant. La *complétude* d'un composant est obtenue par la connexion de tous ses liens à des tableaux du composant de niveau de hiérarchie juste supérieur. La complétude est obtenue par branchement sur des slots. Deux types de complétudes sont :

- *Complétude directe* : la connexion est obtenue par la mise en connexion de tableaux du niveau supérieur avec tous les liens du composant branché. Les deux tableaux ainsi connectés doivent être conformes : même forme, même taille. Le slot de réception est appelé slot direct. L'instanciation du composant ainsi branché aboutit à une exécution unique du code correspondant.
- *Complétude itérative* : les liens du composant connectés sont associés aux tableaux du niveau supérieur par les opérateurs de pavage et d'ajustage. Un des liens est dit maître et permet de définir le domaine d'itération par un recouvrement complet du tableau ainsi pavé. Le composant ainsi connecté est instancié pour un fonctionnement SPMD. Aucun ordre d'exécution n'est spécifié.

Un composant composé est dit complet lorsque tous ses composants sont complets. Les deux types de complétude (directe ou itérative) sont applicables aux deux types de composant (primaire ou composé). Un composant est dit *exécutable* s'il est complet et qu'il ne possède aucun lien d'entrée/sortie. Il devient ainsi une application.

Les tableaux sont multidimensionnels et de type prédéfini : entier, réel ou complexe. Au plus une dimension d'un tableau peut être infinie. Cela permet de prendre en considération le temps dans une des dimensions du tableau, sur laquelle le pavage et l'ajus-

tage restent utilisables. Toutes les dimensions sont toriques, permettant le pavage et l'ajustage sans effet de bord.

Dans GASPARD, les techniques de transformation de code exposées dans le chapitre 3 sont effectivement disponibles. Elles seront utilisées interactivement par le programmeur afin d'adapter le code aux contraintes d'exécution liées à la machine destination : taille mémoire, distribution, localité...

4.3 L'architecture logicielle

Le concept général d'un tel environnement est de proposer un ensemble d'outils (visualisation/édition/transformation) travaillant autour d'une même structure (une application ARRAY-OL). Un certain nombre de ces outils sont déjà en place (une interface graphique, des *parsers*, des *printers*, des générateurs de code, les outils de transformations) mais d'autres pourraient être intégrés ultérieurement. Les objectifs de ces outils n'étant pas les mêmes, ils n'ont pas été écrits dans le même langage. Il était donc souhaitable de prévoir un module dédié à la gestion des structures internes et facilement interfaçable avec les autres pour jouer le rôle de serveur d'information.

4.3.1 Les langages et outils utilisés

D'abord il était important que les langages et bibliothèques choisis existent sur les plates-formes de références c'est à dire essentiellement UNIXTM et WINDOWS^R et qu'ils soient libres. Ce dernier point était essentiel aussi bien pour nous dans le cadre d'une diffusion universitaire que pour TMS dans un cadre industrielle puisque il est prévu que cet atelier fasse partie des contributions commerciales de TMS.

Ensuite les modules autres que l'interface graphique nécessitent surtout du traitement de données et de structures. Ainsi, pour des raisons d'efficacité et de facilité, il a été jugé préférable de conserver le même langage pour tous. Ce langage devait posséder les caractéristiques classiques de structuration du code. Notre choix s'est donc porté sur le langage CAML (en fait OBJECTIF CAML [CMP00]) qui possède les qualités classiques des langages impératifs structurés mais a aussi certaines facilités de codage comme l'existence de structures de tableaux et surtout une gestion automatique de la mémoire (*garbage collector*).

Pour cette version de l'interface graphique, les critères ne sont plus vraiment les mêmes. Le langage devait surtout répondre à des besoins de composants graphiques (menu, arborescence...) et proposer la gestion de leur interactivité. Il a donc été décidé d'utiliser TCL [Rai00]. Il s'agit d'un langage de script ressemblant au C-shell mais dont le grand intérêt concerne son extension TK. Celle-ci offre un certain nombre de composants graphiques ainsi que leur gestion de manière événementielle. Afin de mieux

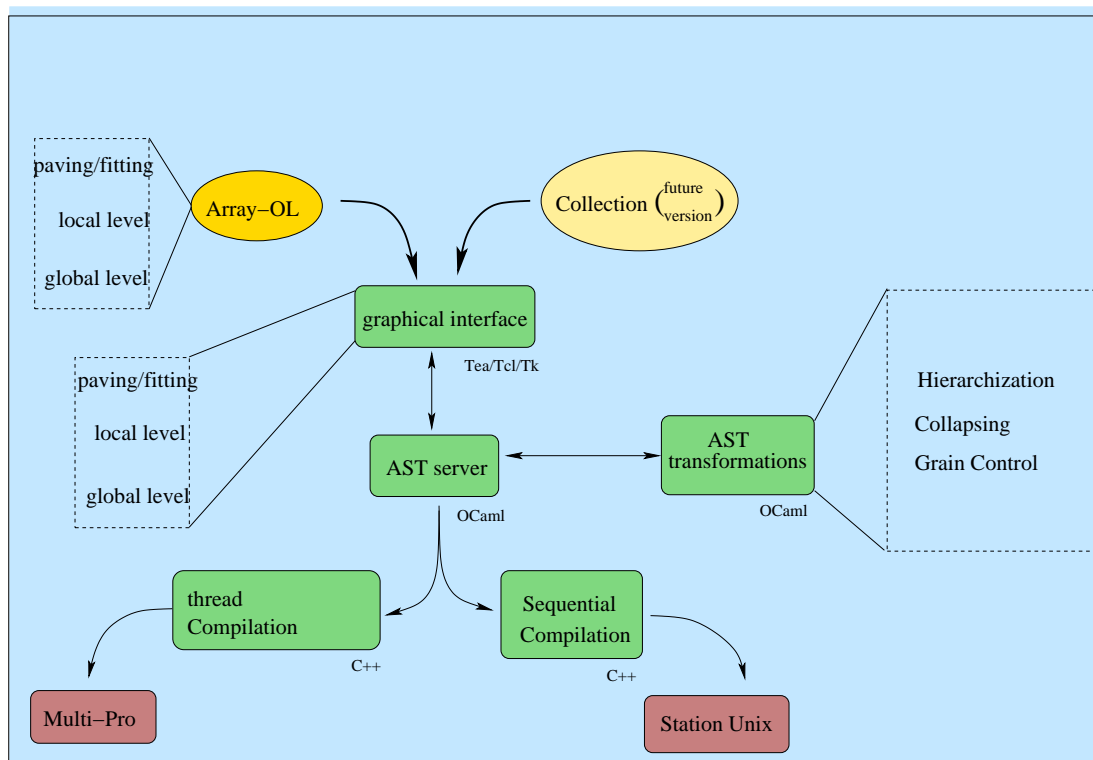


FIG. 4.1 – Architecture de l'environnement GASPARD

structurer le code, nous y avons ajouté la couche TEA qui permet de définir des interfaces objets à la JAVA.

La figure 4.1 représente la construction globale du projet GASPARD [DMS99]. Certaines parties seront explicitées dans la suite de ce document. On retrouve en particulier les opérations de transformation interfacées directement avec le serveur gérant l'arbre syntaxique abstrait.

4.3.2 Arbre syntaxique abstrait

Le cœur déclaratif est constitué de l'arbre syntaxique abstrait (AST) représentant la structure des applications. Celui-ci reprend la plupart des caractéristiques de GASPARD : les tâches sont des slots sur lesquels se branchent des composants. Les différentes entités qui composent l'AST sont :

- `project` : son rôle est de rassembler les éléments (sous-applications et TE) composant une même application TS hiérarchique.
- `compound component` : il correspond au graphe d'un niveau global unique d'une seule application i.e. un étage d'une application TS complète.
- `primary component` : il s'agit exactement des TE.

- `array` : il correspond à un tableau.
- `slot` : il représente une tâche au niveau global en exhibant les dépendances entre les tableaux d'un `compound component`.
- `interactive slot` et `direct slot` : ces deux entités décrivent l'interface c'est-à-dire le passage de paramètres entre un `slot` et le composant qui réalise effectivement la tâche. L'entité `interactive slot` spécifie une interface à la ARRAY-OL (avec déclaration du pavage/ajustage) et l'entité `direct slot` représente simplement le passage des tableaux telles quels au composant.
- `component instance` : il instancie le composant à appeler c'est-à-dire qu'il le nomme et qu'il en fixe les paramètres éventuels.

4.3.3 Le serveur

Les différents modules CAML peuvent accéder directement aux structures internes de l'AST. Pour la partie graphique, il faut prévoir un autre moyen d'interfaçage. Il a été convenu d'utiliser un mode de communication textuel ce qui facilite la connexion avec à peu près tout autre application. Le serveur fonctionne passivement i.e. il se contente de répondre aux questions et aux ordres qu'on lui donne.

Les questions posées au serveur se divisent en deux catégories :

- la gestion de l'AST (création/modification). Les principaux ordres sont `CREATE`, `GET`, `PUT` et `ADD`. Ils permettent respectivement de créer un objet, de lire un champ, de modifier un champ et d'ajouter une valeur dans une table. Les objets sont désignés, soit directement par une clé, soit par un séquence de champs à partir d'un autre objet ayant une clé d'accès. Les valeurs scalaires sont désignées par leur représentation textuelle (figure 4.2);

```

> CREATE array;
@key12;
> PUT @key12.name HYDRO;
HYDRO;
> TYPE @key24;
compound_component;
> ADD @key24.array_set @key12;
arr2;
> GET @key24.slot_set[tache1];
@key98;
> GET @key98.name;
FFT;
```

FIG. 4.2 – Exemples du protocole utilisé pour accéder à l'AST via le serveur

- l'appel de fonction de plus haut niveau dont `FUSION`, `ONE_LEVEL` et `COLLAPSE` pour les transformations, et `GENERATE_CC` et `GENERATE_TAOL` pour générer du code C++ ou du format ARRAY-OL/TMS (figure 4.3).

```

> ONE_LEVEL @key24 { task1 task2 task3 };
new_task { arr1 arr2 };
> GENERATE_CC @key24 "application.cc";
generate_cc
generate_main_header
generate_all_primary_components
...
DONE;

```

FIG. 4.3 – Ordres de transformation de code ARRAY-OL et de génération de code C++

Le fait que ce dialogue avec le serveur soit textuel présente un autre avantage : ce langage de commande est aussi utilisé comme format de sauvegarde (extension `.gas`) i.e. un tel fichier est constitué de l'ensemble des lignes de commande à envoyer au serveur pour recréer l'application. Ainsi on peut être amené à rajouter des informations dans l'AST et continuer à lire les anciens fichiers.

Enfin, CAML étant un langage compilé et se basant sur un typage fort, la conversion des commandes textuelles d'accès à l'AST doit se faire par codage exhaustif sur les types et les champs. Ce travail fastidieux et répétitif est assuré par un générateur automatique.

4.4 Exemple de transformation

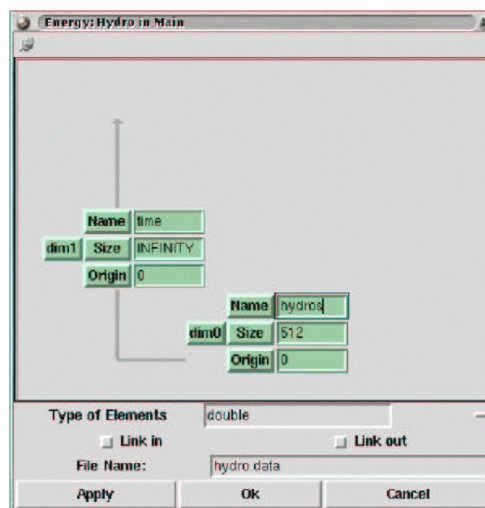
L'exemple reprend celui déjà utilisé au chapitre 3 : la VBL infinie. Nous décrivons ici la construction pas à pas d'une partie de cette application (les trois premières tâches) dans l'environnement GASPARD puis les transformations appliquées au code ainsi obtenu afin de le compiler sur une machine séquentielle avec bien sûr une taille mémoire finie.

► Création des composants

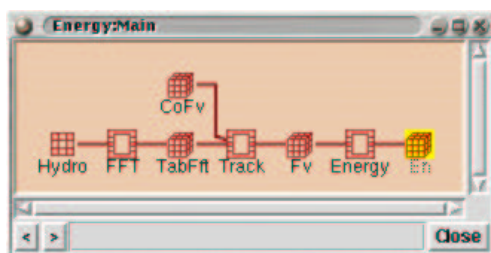
Toute la spécification de cette application peut être faite interactivement dans l'environnement GASPARD. La première étape concerne la création du projet Energy correspondant à cette application, puis de Main le composant de plus haut niveau (cf. figure 4.4(a)). La seconde étape consiste à déclarer les tableaux ainsi que les slots utilisés dans ce composant, et de les relier par une manipulation à la souris afin de spécifier les dépendances entre les données (cf. figures 4.4(b) et 4.4(c)). En même temps, les caractéristiques des tableaux peuvent être saisies (cf. figure 4.4(d)).



(a) Création d'un nouveau projet et de ses composants



(d) Spécification des caractéristiques d'un tableau

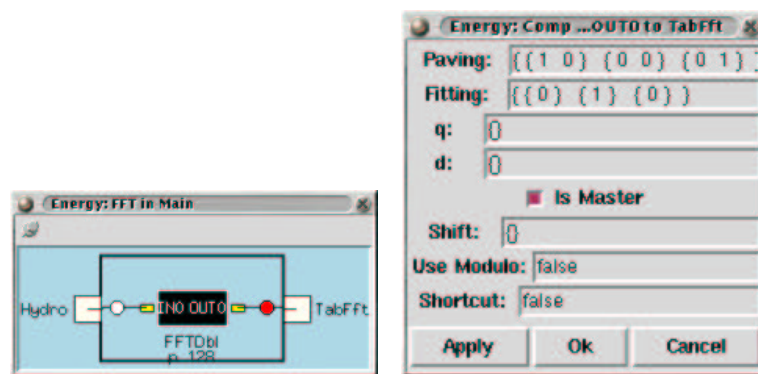


(b) Déclarations et liaisons entre tableaux et slots...



(c) ... par une manipulation de type « drag and drop »

FIG. 4.4 – Déclaration des tableaux et slots pour la VBL infinie



(a) Connexion des tableaux et motifs

(b) Spécification du pavage et ajustage

FIG. 4.5 – Connexion d'un composant primaire dans un slot

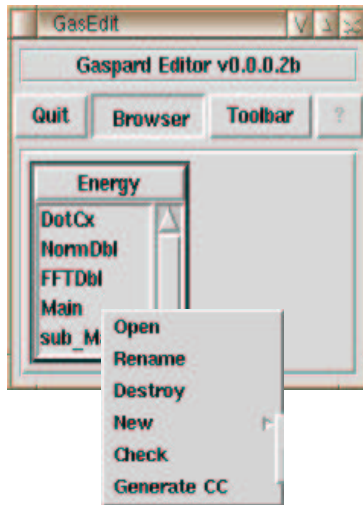
► Spécification des tâches

À cet instant le composant *Main* n'est pas encore exécutable car il reste des slots pour lesquels il n'y a pas de composant de branché. L'utilisateur doit alors connecter des composants créés par lui-même ou extraits d'une bibliothèque ou d'une autre application avant de commencer la génération de code, voire la transformation. Dans cet exemple, nous connectons dans le slot *FFT* par un « drag and drop » le composant primaire *FFTDbl* obtenu depuis une librairie. Chaque slot, une fois connecté, doit être édité afin de spécifier les connexions entre les tableaux en entrée et les liens du composant (cf. figure 4.5(a)). Enfin pour chaque connexion les informations de pavage et ajustage doivent être saisies afin d'obtenir la complétude du composant de niveau supérieur. L'environnement *GASPARD* n'impose aucun ordre dans la façon de saisir les différentes spécifications. Nous proposons néanmoins une approche « top-down » assez intuitive mais non indispensable.

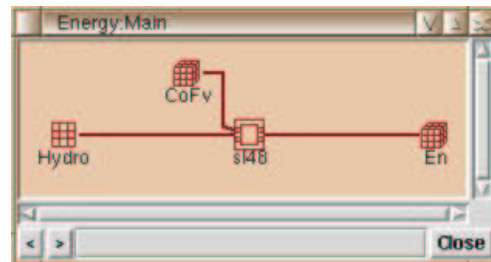
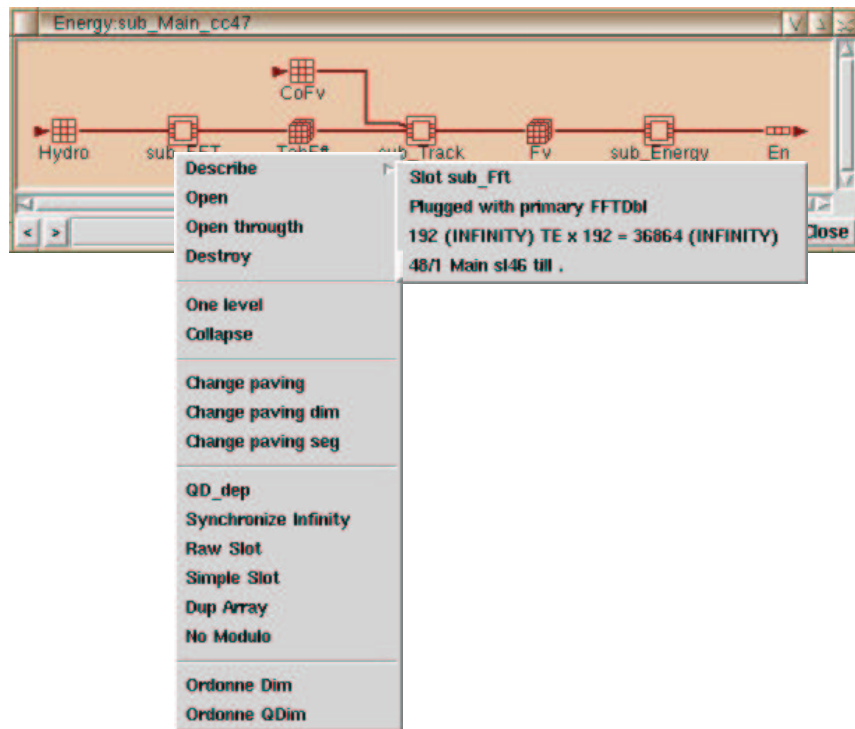
► Transformations et compilation

Une fois l'application complète, celle-ci peut être transformée, compilée puis exécutée. La construction de *GASPARD* autour du serveur d'arbre syntaxique abstrait va permettre de mettre en place les transformations directement sur l'AST et de les visualiser ensuite par l'interface graphique.

Après avoir sélectionné l'ensemble des tâches, on peut appliquer la règle de transformation «one level» (cf. le menu contextuel de la figure 4.6(c)), nous obtenons une nouvelle application avec un niveau de plus dans la hiérarchie. Visuellement la séquence de slots qui concernent des tableaux infinis, est remplacée par un composant



(a) Le nouveau projet

(b) Le composant *Top-level* après hiérarchisation(c) Le composant connecté dans le composant *Top-level*FIG. 4.6 – Hiérarchisation du composant initial *Main* de l'application *Energy*

(b) Le composant *Top-level* calcule sur des tableaux de taille infinie, il ne possède maintenant qu'un seul slot (*sl148*) qui produit directement le tableau infini *En* depuis le tableau infini *Hydro*

(c) Chaque composant, chaque slot et chaque tableau de ce nouveau composant est produit automatiquement par transformations appliquées sur le projet initial.

composé représentant les mêmes tâches à effectuer mais sur des tableaux finis. Cette technique qui aboutie à ne conserver les tableaux infinis qu'au *Top-level* de l'application et connectés à un seul slot, va permettre la génération de code sur une machine (à mémoire finie). On peut vérifier, avec le menu «Describe» sur la première sous-tâche, le taux de recalcul des tâches hiérarchiques (48/1 sur la figure 4.6(c)) et en déduire un agrandissement de la hiérarchie grâce au menu «Change Paving». Finalement, on peut demander la génération du code de l'application par le menu «generate CC» (cf. le menu contextuel de la figure 4.6(a)).

De la même façon, les outils de transformation intégrés à GASPARD sont adaptés à la spécialisation d'applications pour des architectures spécifiques [DDMS99]. La taille des tableaux et donc leur partitionnement et la répartition entre les différents niveaux de hiérarchies mémoires est un point important dans l'obtention de performances.

4.5 Conclusion

Nous avons présenté l'environnement de spécification visuel construit suivant le paradigme data-parallèle (modèle local) dans un graphe de dépendance de données (modèle global) : GASPARD. Celui-ci, dans cette version préliminaire, est fortement dédié aux applications du traitement de signal systématique développées en ARRAY-OL. La notion de composant et la réutilisabilité offrent aux programmeurs un outil de haut niveau pour la définition de ce type d'applications. Le fait de ne pas insérer des directives de placement ou des informations d'ordonnancement laisse libre cours aux compilateurs pour tirer partie au mieux de l'architecture visée. Pour ce faire, les transformations proposées au chapitre 3, permettent au programmeur de transformer les parties de son application par insertion de niveaux de hiérarchie supplémentaires. L'utilisation directe sur des tableaux de taille infinie dans le cas de la VBL reste le seul moyen de pouvoir générer du code à partir d'une unique spécification, indépendante des contraintes matérielles de la machine d'exécution. De la même façon, et à partir de la même spécification, les transformations permettraient une génération de code sur des machines dédiées embarquées.

Conclusion

Les travaux présentés dans cette thèse se situent dans le domaine du traitement de signal. Ces travaux découlent d'une collaboration avec TMS. Cette collaboration concerne principalement la spécification et l'exécution d'une partie du traitement d'une chaîne sonar, d'une chaîne radar, et, par prolongement, d'applications de ce type.

De telles applications combinent une partie d'acquisition et de traitements en temps réel de signaux venant de capteurs puis une partie de traitement et d'analyse des données pour en extraire l'information utile. Nos travaux sont plus particulièrement focalisés sur le traitement de signal intervenant dans ces applications.

Le domaine du traitement de signal possède des caractéristiques bien particulières : traitements réguliers, massivement data-parallèles, linéaires, de faible complexité algorithmique, ayant recours à un jeu réduit de primitives... Cependant, jusqu'à présent, les développements de telles applications sont basés sur des langages classiques et généralistes comme le C, agrémentés de code assembleur. Cette pratique ne tire en rien profit des caractéristiques des applications et produit un code bien souvent très lié à l'architecture cible et difficilement réutilisable. Pour uniformiser l'écriture de ces applications et tirer parti des bonnes propriétés des applications visées, TMS a proposé le langage ARRAY-OL, langage dédié au traitement de signal.

ARRAY-OL se distingue par le choix de tableaux éventuellement infinis comme unique structure de données et des structures de contrôle réduites au strict nécessaire pour l'exécution de tâches complètement data-parallèles. En fait, le langage permet de spécifier l'algorithme de traitement et les dépendances de données sans se soucier des problèmes de placement de données, d'ordonnancement des calculs, ou encore de l'architecture cible.

Nos travaux se situent au niveau de la compilation d'applications de traitement de signal spécifiées en ARRAY-OL. Un support d'exécution du langage ARRAY-OL a été développé et des machines dédiées ARRAY-OL ont été définies chez TMS. Nos travaux de compilation visent une exécution efficace aussi bien sur stations de travail classiques (à des fins de simulation et mise au point des applications), que sur machines dédiées à ARRAY-OL.

La préexistence d'un support d'exécution ARRAY-OL (logiciel ou matériel) nous a conduit à préférer une méthode de compilation par transformation des applications au

niveau du langage plutôt que des stratégies d'implémentation directes. Ainsi la compilation est scindée en deux parties :

- d'une part, nous avons proposé et mis en œuvre un support d'exécution optimisé du langage ARRAY-OL ;
- d'autre part, des transformations d'applications d'ARRAY-OL vers ARRAY-OL, en utilisant la possibilité de hiérarchiser le code, assument le travail de placement et d'ordonnement permettant de satisfaire aux différentes contraintes de mémoires, de latences ou de communications.

Nous avons utilisé un formalisme approprié à la description du langage ARRAY-OL pour mettre en place ces transformations de code et les manipulations que nous envisageons. Ce formalisme des ODT permet de définir les dépendances de points entre les tableaux par la combinaison d'un ensemble d'opérateurs élémentaires. Ils nous ont permis de décrire formellement la transformation de base qui consiste à produire une hiérarchie à partir d'une séquence de deux tâches, puis une série de transformations permettant d'étendre celle-ci à l'ensemble de l'application et de contrôler le processus de hiérarchisation. Cependant le nombre de possibilités de hiérarchies différentes auxquelles peuvent aboutir ces transformations, suivant l'ordre dans lequel on les applique, est d'ordre combinatoire. Nous avons donc défini des mesures objectives afin d'évaluer l'effet de ces transformations et ainsi de guider leur utilisation.

Enfin il était important de proposer ces transformations dans un environnement utilisable par tous et notamment les développeurs d'applications traitement de signal. L'environnement graphique GASPARD a rempli ce rôle en permettant la création, la transformation et enfin la compilation multi-plateformes (séquentielle, SMP, distribuée...) d'applications ARRAY-OL de manière totalement graphique et interactive.

Perspectives

D'un point de vue purement technique, les transformations que nous proposons et particulièrement la fusion qui en est le support, ne sont pas exemptes de limitations. Nous les avons déjà évoquées dans les chapitres précédents. Il s'agit, d'une part, des contraintes qui garantissent la hiérarchisation comme, par exemple, la forme exacte de la partie résultat et les caractéristiques de l'élimination des modules ; d'autre part, du résultat des transformations qui pourrait être amélioré tel que la réduction du tableau intermédiaire ou la prise en compte des recouvrements sans pénaliser la hiérarchisation.

En ce qui concerne les prolongements de nos travaux, un premier objectif qui devrait se concrétiser à court terme, est l'intégration de la chaîne de transformations/compilation dans la nouvelle version de l'environnement GASPARD [BDDM01]. À partir d'une spécification formelle décrite en UML, un arbre syntaxique est obtenu. Celui-ci est rendu accessible par la mise en œuvre d'interfaces spécialisées : génération de code, spécification UML, spécification visuelle (GASPARD). L'une de ces interfaces devrait concerner la mise en pratique des transformations élémentaires sur une appli-

cation traduite en un arbre syntaxique. La finalité de cette intégration concerne à cours terme la mise en place de stratégies de transformations pour diverses cibles d'exécution. Par un mécanisme de mise à jour en direct, toutes les transformations sont propagées soit vers les spécifications formelles, soit vers les spécifications visuelles. Les générateurs de codes sont inchangés par le fait que toute transformation de l'arbre syntaxique produit un nouvel arbre syntaxique. Ce travail est actuellement en cours d'évaluation (début de la thèse de Philippe DUMONT). La justification de ce changement repose essentiellement sur la possibilité d'ouverture des modèles de spécifications en introduisant le traitement de l'irrégulier, avec comme objectif d'étendre le champs des applications au traitement de données intensif.

À l'évidence, et de le cadre de ce nouvel environnement, les transformations que nous avons définies sont des outils dont l'utilisation ne se limite pas à la compilation sur stations de travail. Nos techniques de transformations sont potentiellement utiles à la mise en œuvre d'applications ARRAY-OL sur les architectures cibles plus traditionnelles du traitement de signal : telles que les machines parallèles dédiées, les multiprocesseurs SMP à base de composants « sur l'étagère », les multiprocesseur hétérogènes intégrés « system on chip », voire des systèmes plus ou moins largement distribués. Dans cet esprit, deux applications directes de notre travail ont d'ores et déjà été entreprises. Deux thèses, dont les résultats scientifiques intéressent au plus au point les partenaires du projet ITEA Sophocles, utilisent les transformations élémentaires que nous avons développées.

Le premier projet porte sur la définition d'un environnement d'exécution fortement distribué, dynamique et interactif dédié au traitement de signal. Il constitue le travail de thèse d'Abdelkader AMAR et prévoit la spécification d'applications à partir de graphe de flot de type *data driven* et correspondant au modèle global d'ARRAY-OL. Depuis une identification dynamique de la machine virtuelle distribuée, il convient de transformer, à la compilation mais aussi à l'exécution, la spécification hiérarchique d'une application pour tirer partie au mieux des ressources instantanées d'un réseau dynamique de machines. Nos outils de transformations sont mis en œuvre à partir d'informations produites lors de l'exécution proprement dite de l'application et fournies par le *run-time* lui-même. Cette approche de type *metacomputing* est basée sur un bus CORBA dont les composants peuvent variés interactivement dans le temps. L'exécution de ces applications combine effectivement la distribution de données et de tâches avec des exécutions du type *pipeline* et de type SPMD. Nos travaux apportent les outils indispensables lors des phases de distribution et de redistribution des applications.

Le second projet concerne la compilation et la simulation sur des circuits sur silicium (ou SOC, pour « system on chip »). Il s'agit du thème essentiel du projet Sophocles. De tels systèmes sont constitués d'un assemblage de composants hétérogènes (SIMD, DSP, RISC...) sur un même élément, ou réciproquement de composants virtuels lors de la simulation. Chacun assure les différentes tâches d'un processus de traitement complet de données comme les chaînes sonar ou les relais de communications (acquisition, traitement de signal et traitements de données). Le travail abordé consiste d'abord à spécifier

formellement le type d'architecture du SOC afin de pouvoir explicitement assurer le placement et donc l'ordonnancement d'une application ARRAY-OL. Ces spécifications sont également explicitées en UML et intégrées dans l'environnement GASPARD. Une fois l'application et l'architecture spécifiées, les outils de transformations que nous proposons sont utilisés afin d'assurer un placement efficace de l'application sur le SOC. Là encore les transformations sont code à code et n'interfèrent donc pas avec les différents générateurs de code de chacun des composants. Le reprise des résultats de cette thèse sont un des éléments de référence qui ont suscité le lancement de la thèse de Philippe DUMONT sur ce sujet.

Annexe A

Sources ARRAY-OL des différentes formes de la Veille à Large Bande

A.1 VBL infinie

```
// This file was generated automatically by /usr/local/home/jjss/gaspard/bin/gserv

// -----
// Application Main

Main< Hydro* [( IN_IN0 ) Fft ( OUT_OUT0! )] Tabfft{complex<double>} ;
      Tabfft{complex<double>} Cofv{complex<double>} [( IN_IN0 IN_IN1 )
                                                    F_voies
                                                    ( OUT_OUT0! )]
                                                    Fv{complex<double>} ;
      Fv{complex<double>} [( IN_IN0 ) Energie ( OUT_OUT0! )] Energie* ;
      Energie* [( IN_IN0 ) Rgp_lg_bd ( OUT_OUT0! )] Bl* ;
      Bl* [( IN_IN0 ) Int_cte ( OUT_OUT0! )] Intc* ;
      Intc* Coaz* [( IN_IN1 IN_IN0 ) Stab_az ( OUT_OUT0! )] Stab* ;
      Stab* [( IN_IN0 ) Int_lge ( OUT_OUT0! )] Intl* >

// Array declarations

Main.Hydro ( 512 ~ )
Main.Fv ( 128 200 ~ )
Main.Bl ( 128 ~ )
Main.Tabfft ( 512 256 ~ )
Main.Intl ( 128 ~ )
Main.Coaz ( 128 8 )
Main.Energie ( 128 200 ~ )
Main.Intc ( 128 ~ )
Main.Stab ( 128 ~ )
Main.Cofv ( 128 200 192 )
```

```

// TE reference

Main.Int_lge : Sum1DDbl8(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.Energie : NormDbl(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.Int_cte : Sum1DDbl8(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.F_voies : DotCx(IN_IN0:IN0, IN_IN1:IN1, OUT_OUT0:OUT0)
Main.Fft : FFTDbl(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.Rgp_lg_bd : Sum1DDbl(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.Stab_az : DotDbl8(IN_IN1:IN1, IN_IN0:IN0, OUT_OUT0:OUT0)

// Pattern declarations

Main.Int_lge.IN_IN0 ( 8 )
// Main.Int_lge.OUT_OUT0 : 0-dim array!

// Main.Energie.IN_IN0 : 0-dim array!

// Main.Energie.OUT_OUT0 : 0-dim array!

Main.Int_cte.IN_IN0 ( 8 )
// Main.Int_cte.OUT_OUT0 : 0-dim array!

Main.F_voies.IN_IN0 ( 192 )
Main.F_voies.IN_IN1 ( 192 )
// Main.F_voies.OUT_OUT0 : 0-dim array!

Main.Fft.IN_IN0 ( 512 )
Main.Fft.OUT_OUT0 ( 256 )
Main.Rgp_lg_bd.IN_IN0 ( 200 )
// Main.Rgp_lg_bd.OUT_OUT0 : 0-dim array!

Main.Stab_az.IN_IN1 ( 8 )
Main.Stab_az.IN_IN0 ( 8 )
// Main.Stab_az.OUT_OUT0 : 0-dim array!

// Task attributes

Main.Int_lge/$Q = ( 128 ~ )
Main.Int_lge.IN_IN0/$A = ( 0 ; 1 ; )
Main.Int_lge.IN_IN0/$P = ( 1 0 ; 0 8 ; )
Main.Int_lge.IN_IN0/$O = ( 0 0 )
// Main.Int_lge.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.Int_lge.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.Int_lge.OUT_OUT0/$O = ( 0 0 )
Main.Energie/$Q = ( 128 200 ~ )
// Main.Energie.IN_IN0/$A = : empty array or array of empty arrays!

Main.Energie.IN_IN0/$P = ( 1 0 0 ; 0 1 0 ; 0 0 1 ; )

```

```

Main.Energie.IN_IN0/$O = ( 0 0 0 )
// Main.Energie.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.Energie.OUT_OUT0/$P = ( 1 0 0 ; 0 1 0 ; 0 0 1 ; )
Main.Energie.OUT_OUT0/$O = ( 0 0 0 )
Main.Int_cte/$Q = ( 128 ~ )
Main.Int_cte.IN_IN0/$A = ( 0 ; 1 ; )
Main.Int_cte.IN_IN0/$P = ( 1 0 ; 0 8 ; )
Main.Int_cte.IN_IN0/$O = ( 0 0 )
// Main.Int_cte.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.Int_cte.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.Int_cte.OUT_OUT0/$O = ( 0 0 )
Main.F_voies/$Q = ( 128 200 ~ )
Main.F_voies.IN_IN0/$A = ( 1 ; 0 ; 0 ; )
Main.F_voies.IN_IN0/$P = ( 4 0 0 ; 0 1 0 ; 0 0 1 ; )
Main.F_voies.IN_IN0/$O = ( 0 28 0 )
Main.F_voies.IN_IN1/$A = ( 0 ; 0 ; 1 ; )
Main.F_voies.IN_IN1/$P = ( 1 0 0 ; 0 1 0 ; 0 0 0 ; )
Main.F_voies.IN_IN1/$O = ( 0 0 0 )
// Main.F_voies.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.F_voies.OUT_OUT0/$P = ( 1 0 0 ; 0 1 0 ; 0 0 1 ; )
Main.F_voies.OUT_OUT0/$O = ( 0 0 0 )
Main.Fft/$Q = ( 512 ~ )
Main.Fft.IN_IN0/$A = ( 0 ; 1 ; )
Main.Fft.IN_IN0/$P = ( 1 0 ; 0 512 ; )
Main.Fft.IN_IN0/$O = ( 0 0 )
Main.Fft.OUT_OUT0/$A = ( 0 ; 1 ; 0 ; )
Main.Fft.OUT_OUT0/$P = ( 1 0 ; 0 0 ; 0 1 ; )
Main.Fft.OUT_OUT0/$O = ( 0 0 0 )
Main.Rgp_lg_bd/$Q = ( 128 ~ )
Main.Rgp_lg_bd.IN_IN0/$A = ( 0 ; 1 ; 0 ; )
Main.Rgp_lg_bd.IN_IN0/$P = ( 1 0 ; 0 0 ; 0 1 ; )
Main.Rgp_lg_bd.IN_IN0/$O = ( 0 0 0 )
// Main.Rgp_lg_bd.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.Rgp_lg_bd.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.Rgp_lg_bd.OUT_OUT0/$O = ( 0 0 )
Main.Stab_az/$Q = ( 128 ~ )
Main.Stab_az.IN_IN1/$A = ( 1 ; 0 ; )
Main.Stab_az.IN_IN1/$P = ( 1 0 ; 0 1 ; )
Main.Stab_az.IN_IN1/$O = ( 0 0 )
Main.Stab_az.IN_IN0/$A = ( 0 ; 1 ; )
Main.Stab_az.IN_IN0/$P = ( 1 0 ; 0 0 ; )
Main.Stab_az.IN_IN0/$O = ( 0 0 )
// Main.Stab_az.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.Stab_az.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.Stab_az.OUT_OUT0/$O = ( 0 0 )

```

```
// End of file
```

A.2 VBL infinie hiérarchisée

```
// This file was generated automatically by /usr/local/home/jjss/gaspard/bin/gserv
```

```
// -----
// Application Main

Main< Hydro* Cofv{complex<double>} Coaz* [( IN_Hydro IN_Cofv IN_Coaz )
s154
( OUT_Intl! )] Intl* >

// Array declarations

Main.Cofv ( 128 200 192 )
Main.Coaz ( 128 8 )
Main.Intl ( 128 ~ )
Main.Hydro ( 512 ~ )

// TE reference

// Pattern declarations

Main.s154.IN_Hydro ( 64 512 512 )
Main.s154.IN_Cofv ( 64 200 128 192 )
Main.s154.IN_Coaz ( 128 8 8 )
Main.s154.OUT_Intl ( 128 )

// Task attributes

Main.s154/$Q = ( ~ )
Main.s154.IN_Hydro/$A = ( 0 1 0 ; 512 0 1 ; )
Main.s154.IN_Hydro/$P = ( 0 ; 32768 ; )
Main.s154.IN_Hydro/$O = ( 0 0 )
Main.s154.IN_Cofv/$A = ( 0 0 1 0 ; 0 1 0 0 ; 0 0 0 1 ; )
Main.s154.IN_Cofv/$P = ( 0 ; 0 ; 0 ; )
Main.s154.IN_Cofv/$O = ( 0 0 0 )
Main.s154.IN_Coaz/$A = ( 1 0 0 ; 0 0 1 ; )
Main.s154.IN_Coaz/$P = ( 0 ; 0 ; )
Main.s154.IN_Coaz/$O = ( 0 0 )
Main.s154.OUT_Intl/$A = ( 1 ; 0 ; )
Main.s154.OUT_Intl/$P = ( 0 ; 1 ; )
Main.s154.OUT_Intl/$O = ( 0 0 )

// -----
```

```

// Application Main.sl54

Main.sl54< Hydro* Cofv{complex<double>} [( IN_Hydro IN_Cofv )
                                         sl58
                                         ( OUT_Energie! )]
      Energie* ;
      Energie* [( IN_IN0 ) sub_Rgp_lg_bd ( OUT_OUT0! )] Bl* ;
      Bl* [( IN_IN0 ) sub_Int_cte ( OUT_OUT0! )] Intc* ;
      Coaz* Intc* [( IN_IN0 IN_IN1 ) sub_Stab_az ( OUT_OUT0! )] Stab* ;
      Stab* [( IN_IN0 ) sub_Int_lge ( OUT_OUT0! )] Intl* >

// Array declarations

Main.sl54.Cofv ( 64 200 128 192 )
Main.sl54.Coaz ( 128 8 8 )
Main.sl54.Hydro ( 64 512 512 )
Main.sl54.Intl ( 128 )
Main.sl54.Stab ( 128 8 )
Main.sl54.Intc ( 128 8 )
Main.sl54.Bl ( 128 64 )
Main.sl54.Energie ( 128 200 64 )

// TE reference

Main.sl54.sub_Rgp_lg_bd : Sum1DDbl(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.sl54.sub_Int_cte : Sum1DDbl8(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.sl54.sub_Stab_az : DotDbl8(IN_IN0:IN0, IN_IN1:IN1, OUT_OUT0:OUT0)
Main.sl54.sub_Int_lge : Sum1DDbl8(IN_IN0:IN0, OUT_OUT0:OUT0)

// Pattern declarations

Main.sl54.sub_Rgp_lg_bd.IN_IN0 ( 200 )
// Main.sl54.sub_Rgp_lg_bd.OUT_OUT0 : 0-dim array!

Main.sl54.sub_Int_cte.IN_IN0 ( 8 )
// Main.sl54.sub_Int_cte.OUT_OUT0 : 0-dim array!

Main.sl54.sub_Stab_az.IN_IN0 ( 8 )
Main.sl54.sub_Stab_az.IN_IN1 ( 8 )
// Main.sl54.sub_Stab_az.OUT_OUT0 : 0-dim array!

Main.sl54.sub_Int_lge.IN_IN0 ( 8 )
// Main.sl54.sub_Int_lge.OUT_OUT0 : 0-dim array!

Main.sl54.sl58.IN_Hydro ( 512 512 )
Main.sl54.sl58.IN_Cofv ( 200 128 192 )
Main.sl54.sl58.OUT_Energie ( 200 128 )

// Task attributes

Main.sl54.sub_Rgp_lg_bd/$Q = ( 128 64 )

```



```

                                                                    (OUT_OUT0! )]
Fv{complex<double>} ;
Fv{complex<double>} [( IN_IN0 )
sub_sub_Energie
( OUT_OUT0! )] Energie* >

// Array declarations

Main.sl54.sl58.Cofv ( 200 128 192 )
Main.sl54.sl58.Tabfft ( 512 256 1 )
Main.sl54.sl58.Hydro ( 512 512 )
Main.sl54.sl58.Energie ( 200 128 )
Main.sl54.sl58.Fv ( 128 200 1 )

// TE reference

Main.sl54.sl58.sub_sub_Fft : FFTDb1(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.sl54.sl58.sub_sub_F_voies : DotCx(IN_IN1:IN1,
IN_IN0:IN0,
OUT_OUT0:OUT0)
Main.sl54.sl58.sub_sub_Energie : NormDb1(IN_IN0:IN0, OUT_OUT0:OUT0)

// Pattern declarations

Main.sl54.sl58.sub_sub_Fft.IN_IN0 ( 512 )
Main.sl54.sl58.sub_sub_Fft.OUT_OUT0 ( 256 )
Main.sl54.sl58.sub_sub_F_voies.IN_IN1 ( 192 )
Main.sl54.sl58.sub_sub_F_voies.IN_IN0 ( 192 )
// Main.sl54.sl58.sub_sub_F_voies.OUT_OUT0 : 0-dim array!

// Main.sl54.sl58.sub_sub_Energie.IN_IN0 : 0-dim array!

// Main.sl54.sl58.sub_sub_Energie.OUT_OUT0 : 0-dim array!

// Task attributes

Main.sl54.sl58.sub_sub_Fft/$Q = ( 512 )
Main.sl54.sl58.sub_sub_Fft.IN_IN0/$A = ( 0 ; 1 ; )
Main.sl54.sl58.sub_sub_Fft.IN_IN0/$P = ( 1 ; 0 ; )
Main.sl54.sl58.sub_sub_Fft.IN_IN0/$O = ( 0 0 )
Main.sl54.sl58.sub_sub_Fft.OUT_OUT0/$A = ( 0 ; 1 ; 0 ; )
Main.sl54.sl58.sub_sub_Fft.OUT_OUT0/$P = ( 1 ; 0 ; 0 ; )
Main.sl54.sl58.sub_sub_Fft.OUT_OUT0/$O = ( 0 0 0 )
Main.sl54.sl58.sub_sub_F_voies/$Q = ( 200 128 )
Main.sl54.sl58.sub_sub_F_voies.IN_IN1/$A = ( 0 ; 0 ; 1 ; )
Main.sl54.sl58.sub_sub_F_voies.IN_IN1/$P = ( 1 0 ; 0 1 ; 0 0 ; )
Main.sl54.sl58.sub_sub_F_voies.IN_IN1/$O = ( 0 0 0 )
Main.sl54.sl58.sub_sub_F_voies.IN_IN0/$A = ( 1 ; 0 ; 0 ; )
Main.sl54.sl58.sub_sub_F_voies.IN_IN0/$P = ( 0 4 ; 1 0 ; 0 0 ; )
Main.sl54.sl58.sub_sub_F_voies.IN_IN0/$O = ( 0 28 0 )

```

```
// Main.sl54.sl58.sub_sub_F_voies.OUT_OUT0/$A = : empty array or array of empty arrays

Main.sl54.sl58.sub_sub_F_voies.OUT_OUT0/$P = ( 0 1 ; 1 0 ; 0 0 ; )
Main.sl54.sl58.sub_sub_F_voies.OUT_OUT0/$O = ( 0 0 0 )
Main.sl54.sl58.sub_sub_Energie/$Q = ( 200 128 )
// Main.sl54.sl58.sub_sub_Energie.IN_IN0/$A = : empty array or array of empty arrays!

Main.sl54.sl58.sub_sub_Energie.IN_IN0/$P = ( 0 1 ; 1 0 ; 0 0 ; )
Main.sl54.sl58.sub_sub_Energie.IN_IN0/$O = ( 0 0 0 )
// Main.sl54.sl58.sub_sub_Energie.OUT_OUT0/$A = : empty array or array of empty arrays

Main.sl54.sl58.sub_sub_Energie.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.sl54.sl58.sub_sub_Energie.OUT_OUT0/$O = ( 0 0 )

// End of file
```

A.3 VBL utilisée pour les tests de performance

```
// This file was generated automatically by /usr/local/home/jjss/gaspard/bin/gserv

// -----
// Application Main

Main< Cofv{complex<double>} Hydro* [( IN_Cofv IN_Hydro )
                                sl89
                                ( OUT_Energie! )] Energie* >

// Array declarations

Main.Hydro ( 512 4096 )
Main.Energie ( 128 200 8 )
Main.Cofv ( 128 200 192 )

// TE reference

// Pattern declarations

Main.sl89.IN_Cofv ( 200 8 192 )
Main.sl89.IN_Hydro ( 220 512 )
Main.sl89.OUT_Energie ( 200 8 )

// Task attributes

Main.sl89/$Q = ( 16 8 )
Main.sl89.IN_Cofv/$A = ( 0 1 0 ; 1 0 0 ; 0 0 1 ; )
Main.sl89.IN_Cofv/$P = ( 8 0 ; 0 0 ; 0 0 ; )
```

```

Main.sl89.IN_Cofv/$O = ( 0 0 0 )
Main.sl89.IN_Hydro/$A = ( 1 0 ; 0 1 ; )
Main.sl89.IN_Hydro/$P = ( 32 0 ; 0 512 ; )
Main.sl89.IN_Hydro/$O = ( 0 0 )
Main.sl89.OUT_Energie/$A = ( 0 1 ; 1 0 ; 0 0 ; )
Main.sl89.OUT_Energie/$P = ( 8 0 ; 0 0 ; 0 1 ; )
Main.sl89.OUT_Energie/$O = ( 0 0 0 )

// -----
// Application Main.sl89

Main.sl89< Hydro* [( IN_IN0 ) sub_Fft ( OUT_OUT0! )] Tabfft{complex<double>} ;
      Tabfft{complex<double>} Cofv{complex<double>} [( IN_IN0 IN_IN1 )
      sub_F_voies
      ( OUT_OUT0! )]
      Fv{complex<double>} ;
      Fv{complex<double>} [( IN_IN0 ) sub_Energie ( OUT_OUT0! )] Energie* >

// Array declarations

Main.sl89.Fv ( 8 200 1 )
Main.sl89.Energie ( 200 8 )
Main.sl89.Hydro ( 220 512 )
Main.sl89.Tabfft ( 220 256 1 )
Main.sl89.Cofv ( 200 8 192 )

// TE reference

Main.sl89.sub_Energie : NormDbl(IN_IN0:IN0, OUT_OUT0:OUT0)
Main.sl89.sub_F_voies : DotCx(IN_IN0:IN0, IN_IN1:IN1, OUT_OUT0:OUT0)
Main.sl89.sub_Fft : FFTDbl(IN_IN0:IN0, OUT_OUT0:OUT0)

// Pattern declarations

// Main.sl89.sub_Energie.IN_IN0 : 0-dim array!

// Main.sl89.sub_Energie.OUT_OUT0 : 0-dim array!

Main.sl89.sub_F_voies.IN_IN0 ( p )
Main.sl89.sub_F_voies.IN_IN1 ( p )
// Main.sl89.sub_F_voies.OUT_OUT0 : 0-dim array!

Main.sl89.sub_Fft.IN_IN0 ( 512 )
Main.sl89.sub_Fft.OUT_OUT0 ( 256 )

// Task attributes

Main.sl89.sub_Energie/$Q = ( 200 8 )
// Main.sl89.sub_Energie.IN_IN0/$A = : empty array or array of empty arrays!

Main.sl89.sub_Energie.IN_IN0/$P = ( 0 1 ; 1 0 ; 0 0 ; )

```

```

Main.sl89.sub_Energie.IN_IN0/$O = ( 0 0 0 )
// Main.sl89.sub_Energie.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.sl89.sub_Energie.OUT_OUT0/$P = ( 1 0 ; 0 1 ; )
Main.sl89.sub_Energie.OUT_OUT0/$O = ( 0 0 )
Main.sl89.sub_F_voies/$Q = ( 200 8 )
Main.sl89.sub_F_voies.IN_IN0/$A = ( 1 ; 0 ; 0 ; )
Main.sl89.sub_F_voies.IN_IN0/$P = ( 0 4 ; 1 0 ; 0 0 ; )
Main.sl89.sub_F_voies.IN_IN0/$O = ( 0 28 0 )
Main.sl89.sub_F_voies.IN_IN1/$A = ( 0 ; 0 ; 1 ; )
Main.sl89.sub_F_voies.IN_IN1/$P = ( 1 0 ; 0 1 ; 0 0 ; )
Main.sl89.sub_F_voies.IN_IN1/$O = ( 0 0 0 )
// Main.sl89.sub_F_voies.OUT_OUT0/$A = : empty array or array of empty arrays!

Main.sl89.sub_F_voies.OUT_OUT0/$P = ( 0 1 ; 1 0 ; 0 0 ; )
Main.sl89.sub_F_voies.OUT_OUT0/$O = ( 0 0 0 )
Main.sl89.sub_Fft/$Q = ( 220 )
Main.sl89.sub_Fft.IN_IN0/$A = ( 0 ; 1 ; )
Main.sl89.sub_Fft.IN_IN0/$P = ( 1 ; 0 ; )
Main.sl89.sub_Fft.IN_IN0/$O = ( 0 0 )
Main.sl89.sub_Fft.OUT_OUT0/$A = ( 0 ; 1 ; 0 ; )
Main.sl89.sub_Fft.OUT_OUT0/$P = ( 1 ; 0 ; 0 ; )
Main.sl89.sub_Fft.OUT_OUT0/$O = ( 0 0 0 )

// End of file

```

Bibliographie personnelle

- [BDL⁺01] Pierre Boulet, Jean-Luc Dekeyser, Jean-Luc Levaire, Philippe Marquet, Julien Soula, and Alain Demeure. Visual data-parallel programming for signal processing applications. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, février 2001. IEEE Computer Society Press.
- [DDMS99] Jean-Luc Dekeyser, Alain Demeure, Philippe Marquet, and Julien Soula. Compilation Array-OL. Rapport final 1998-1999, LIFL, Université de Lille et Thomson Marconi Sonar, Sophia-Antipolis, mai 1999. 69 pages.
- [DMS99] Jean-Luc Dekeyser, Philippe Marquet, and Julien Soula. Video kills the radio stars. In *Supercomputing'99 (poster session)*, Portland, OR, novembre 1999.
- [SMDD01] Julien Soula, Philippe Marquet, Alain Demeure, and Jean-Luc Dekeyser. Compilation principle of a specification language dedicated to signal processing. In *Parallel Computing Technologies (PaCT 2001)*, Novosibirsk, Russia, septembre 2001. Lecture Notes in Computer Science.

Bibliographie

- [ABG⁺97] C. Ancourt, D. Barthou, C. Guettier, F. Irigoien, B. Jourdan, and J. Mattioli. Automatic data mapping of signal processing applications. In *Application Spec. Array Processors*, pages 350–362, Zurich, Switzerland, juillet 1997.
- [ABM⁺92] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. New York : McGraw-Hill, 1992.
- [AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, octobre 2001.
- [Ame83] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*. Springer-Verlag, 1983.
- [AP93] C. André and M. Peraldi. Grafcet and synchronous languages. *Revue RAIRO Automatique*, 27(1), 1993.
- [BCD⁺89] P. Borrás, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR : The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.
- [BDD⁺99] Pierre Boulet, Jean-Luc Dekeyser, Alain Demeure, Florent Devin, and Philippe Marquet. Une approche à la SQL du traitement de données intensif dans Gaspard. In *RenPar'11, Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 151–156, Rennes, juin 1999.
- [BDDM01] Pierre Boulet, Jean-Luc Dekeyser, Florent Devin, and Philippe Marquet. A visual development environment for meta-computing applications. In *HCI International 2001, 9th Int'l Conf. on Human-Computer Interaction*, New Orleans, LA, août 2001. Lawrence Erlbaum Associates, Publishers.
- [BDL⁺01] Pierre Boulet, Jean-Luc Dekeyser, Jean-Luc Levaire, Philippe Marquet, Julien Soula, and Alain Demeure. Visual data-parallel programming for signal processing applications. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, février 2001. IEEE Computer Society Press.

- [BDSV97] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms : from parallelism extraction to code generation. Rapport de recherche 97-17, LIP, ENS-Lyon, France, juin 1997.
- [BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.
- [Bou91] Luc Bougé. On the semantics of languages for massively parallel SIMD architectures. In *Proc. of the Conf. on Parallel Architecture and Languages Europe (PARLE '91)*, Eindhoven, The Netherlands, juin 1991.
- [Bou93] Luc Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *Technique et Science Informatiques*, 12(6), 1993.
- [Bou96] P. Boulet. Bouclettes : A Fortran loop parallelizer. *Lecture Notes in Computer Science*, 1067, 1996.
- [BR99] Pierre Boulet and Xavier Redon. SPPoC : Symbolic parameterized polyhedral calculator. In *Workshop Compilation et Parallélisation Automatique*, St Nabor, France, octobre 1999.
- [CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly & Associates, avril 2000.
- [CSP90] P. Couronné, J. Saint, and J. Plaice. The Esterel-Lustre OC portable format. Rapport de recherche, Ecoles des mines / INRIA, Sophia-Antipolis, France, 1990.
- [Dav00] J. Davis. Ptolemy II - heterogeneous concurrent modeling and design in Java. Rapport technique, University of California at Berkeley, septembre 2000.
- [Dax97] A. Dax. An elementary proof of Farkas' lemma. *SIAM Review*, 39(3) :503–507, 1997.
- [DDMS99] Jean-Luc Dekeyser, Alain Demeure, Philippe Marquet, and Julien Soula. Compilation Array-OL. Rapport final 1998-1999, LIFL, Université de Lille et Thomson Marconi Sonar, Sophia-Antipolis, mai 1999. 69 pages.
- [Dem98] Alain Demeure. Les ODT : propositions de notation pour décrire des opérateurs de distribution de tableaux. Rapport technique, Thomson Marconi Sonar, Sophia-Antipolis, France, 1998.
- [DLB⁺95] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, septembre 1995.
- [DMS99] Jean-Luc Dekeyser, Philippe Marquet, and Julien Soula. Video kills the radio stars. In *Supercomputing'99 (poster session)*, Portland, OR, novembre 1999.

- [Fea88] Paul Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22 :243–268, septembre 1988.
- [Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.
- [Gen97] Stéphane Genaud. *Transformations de Programmes PEI : Applications au Parallélisme de Données*. Thèse de doctorat, Université Louis Pasteur, Strasbourg, France, 1997.
- [GS66] S. Ginsburg and E. Spanier. Semi-groups, Pressburger formulas and languages. *Pacific Journal of Mathematics*, 16 :285–296, 1966.
- [HAA⁺96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29, 1996.
- [Har87] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, juin 1987.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proc. of the IEEE*, 79(9) :1305–1320, septembre 1991.
- [HPF97] HPF Forum. "*HPF Language Spécification*", janvier 1997.
- [IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization : An overview of the PIPS project. In *1991 International Conference on Supercomputing*, Cologne, juin 1991.
- [JBM94] E.A. Lee J.T. Buck, S. Ha and D.G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4 :155–182, avril 1994.
- [JG88] Geraint Jones and Michael Goldsmith. *Programming in Occam 2*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [KMP⁺96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Speisman, and David Wonnacott. The Omega Library Interface Guide. Rapport de recherche, dept. of Computer Science, 1996.
- [Kok96] Boris Kokoszko. Intégration du modèle data-parallèle irrégulier Idole dans C++. In *Renpar8, 8es Rencontres sur le Parallélisme*, Bordeaux, France, mai 1996.
- [KP93a] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Rapport de recherche, Dept. of Computer Science, University of Maryland, College Park, avril 1993.
- [KP93b] Ronan Keryell and Nicolas Paris. Activity counter : New optimization for the dynamic scheduling of SIMD control flow. In *Int'l Conf. on Parallel Processing*, St. Charles, IL, août 1993.

- [Li01] Chen Li. *Exact Geometric Computation : Theory and Applications*. PhD thesis, NYU, janvier 2001. <http://cs.nyu.edu/exact/doc/>.
- [LP95] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, mai 1995.
- [Mar93] Philippe Marquet. Langages et expression du parallélisme de données. *Technique et Science Informatiques*, 12(6) :685–714, 1993.
- [Mat94] The MathWorks Inc. *MATLAB Reference Guide*, 1994.
- [Mau89] Christophe Mauras. *Alpha : un Langage Équationnel pour la Conception et la Programmation d'Architectures Synchrones*. Thèse de doctorat, Université de Rennes, décembre 1989.
- [MNP+96] Agostino Mathis, Paolo Novelli, Paolo Palazzari, Francesco Romanelli, Corrado Ronchi, and Vittorio Rosato. *HPCN at ENEA*. Relations Central Function, octobre 1996. <http://www.enea.it/hpcn/moshpce/hpcn01e.html>.
- [MP99a] A. Marongiu and P. Palazzari. Mapping system of affine recurrence equations (sare) onto distributed memory systems. In *International Parallel Processing Symposium (IPPS99)*, San Juan, Puerto Rico, avril 1999.
- [MP99b] A. Marongiu and P. Palazzari. Optimization of automatically generated parallel programs. In *The 3rd IMACS International Multiconference on Circuits, Systems, Communications and Computers (CSCC'99)*, Athens, Greece, juillet 1999.
- [MPCM00] Alessandro Marongiu, Paolo Palazzari, Luigi Cinque, and Ferdinando Mastronardo. High level software synthesis of affine iterative algorithms onto parallel architectures. In *HPCN*, volume 1823, pages 333–342. Springer, mai 2000.
- [MPI94] MPI : A message-passing interface standard. Rapport technique UT-CS-94-230, Department of Computer Science at the University of Tennessee, Knoxville, TN, 1994.
- [Pou99] Thierry Poupart. *Compilation de langage data-parallèle sur machine multiprocesseur multithreadée*. Mémoire de DEA, Laboratoire d'informatique fondamentale de Lille, Université de Lille 1, juin 1999.
- [PTMC91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, 1991.
- [Pug92] William Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, volume 8, pages 102–114, août 1992.
- [Rai00] Paul Raines. *"Tcl/Tk"*. O'Reilly & Associates, avril 2000.
- [Ros87] John Rose. C* : A C++-like language for data-parallel computation. In *Proc. USENIX C++ Conf.*, pages 127–134, Santa Fe, NM, décembre 1987.

- [SG95] N. Sundaresan and D. Gannon. Coir : A thread model for supporting task and data parallelism in object-oriented parallel languages. Rapport de recherche 429, Computer Science Department, Indiana University, 1995.
- [SMDD01] Julien Soula, Philippe Marquet, Alain Demeure, and Jean-Luc Dekeyser. Compilation principle of a specification language dedicated to signal processing. In *Parallel Computing Technologies (PaCT 2001)*, volume 2127, pages 358–370, Novosibirsk, Russia, septembre 2001. Lecture Notes in Computer Science.
- [Sun88] Sun Microsystems. RPC : Remote procedure call protocol specification version 2 ; RFC1058. *Internet Request for Comments*, (1057), 1988.
- [Sun90] V. S. Sunderam. PVM : a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4) :315–340, 1990.
- [UJ98] M. Uzam and A. H. Jones. Discrete event control system design using automation petri nets and their ladder diagram implementation. *International Journal of Advanced Manufacturing Systems, special issue on Petri Nets Applications in Manufacturing Systems*, 14(10) :716–728, octobre 1998.
- [UZCZ97] M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zimaan. Vienna Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10) :1068–1083, octobre 1997.
- [VMQ91] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3) :173–182, septembre 1991.
- [VP92] E. Violard and G. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18 :1167–1184, 1992.
- [Wil93] D.K. Wilde. A library for doing polyhedral operations. Rapport de recherche 785, IRISA, Rennes, France, décembre 1993.

Résumé Les applications de traitements de signal (TS), qu'on trouve notamment dans les chaînes sonar, ont des caractéristiques algorithmiques bien particulières. Afin de répondre aux besoins de spécification et de standardisation de celles-ci, TMS (Thomson Marconi Sonar) a développé un langage orienté TS : ARRAY-OL (Array Oriented Language). Il permet de spécifier l'algorithme de calcul et les dépendances de données sans se soucier des problèmes de placement et d'ordonnement.

Nos travaux se situent au niveau de la compilation d'applications spécifiées en ARRAY-OL visant autant les stations de travail classiques (pour la simulation) que des machines dédiées à ARRAY-OL. La préexistence d'un support d'exécution ARRAY-OL (logiciel et matériel) nous a conduit à préférer une méthode de compilation par transformation des applications au niveau du langage (introduction de niveaux hiérarchiques) plutôt que des stratégies d'implémentation directes.

Pour mettre en place ces transformations, nous avons utilisé un formalisme approprié à la description du langage ARRAY-OL : les ODT (Opérateurs de Distribution de Tableaux). Ils nous ont permis de décrire formellement les transformations qui consiste à produire une ou plusieurs hiérarchies à partir d'une séquence de tâches et de contrôler le grain de celles-ci. Devant le nombre de schémas différents que ces transformations peuvent engendrer, nous avons également défini des mesures permettant d'évaluer l'effet de ces transformations afin de guider leur utilisation.

Enfin l'environnement graphique GASPARD rend accessible ces outils à tous, et notamment aux développeurs d'applications TS, en permettant la création, la transformation et la compilation multi-plateformes (séquentielle, SMP, distribuée...) d'applications ARRAY-OL de manière totalement graphique et interactive.

Mots-clés Traitement de signal, parallélisme de données, compilation, transformation de code

Abstract The applications of signal processing (SP), like sonar processing chains, have quite particular algorithmic characteristics. In order to standardize the specification of these applications, TMS (Thomson Marconi Sonar) has developed a SP oriented language: ARRAY-OL (Array Oriented Language). It allows to specify the computation algorithm and the data dependences without worrying about mapping or scheduling.

We have focused on the compilation of applications specified in ARRAY-OL aiming as much traditional workstations (for simulation) as ARRAY-OL dedicated systems. Owing to the pre-existence of a support of execution for ARRAY-OL (software and hardware), we have preferred a compilation method that transform applications at the level of the language (by introduction of hierarchical levels) rather than a strategy of direct implementation.

In order to set up these transformations, we have used a formalism suited to the description of ARRAY-OL: the ODT (*Opérateurs de Distribution de Tableaux*, Array Distributions Operators in english). ODT let us formally describe the transformations which consists in producing one or more hierarchies from a sequence of tasks and to control their granularity. Giving the number of different schema these transformations can generate, we have also defined measurements which allow to evaluate the effect of these transformations in order to guide their use.

Finally the graphical environment GASPARD makes available these tools to everyone, and in particular to the developers of SP applications, by allowing users to create, transform and compile ARRAY-OL applications toward multi-platforms (sequential, SMP, distributed...) by a completely graphic and interactive way.

Keywords Signal processing, data-parallelism, compilation, code transformation