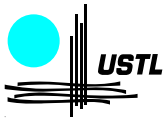


Numéro d'ordre : 3756

Université des Sciences et Technologies de Lille

Thèse

présentée pour obtenir le grade de docteur
spécialité : Informatique
par

Philippe DUMONT

Spécification multidimensionnelle pour le traitement du signal systématique

Thèse soutenue le 15 decembre 2005, devant la commission d'examen formée de :

MM.	El-Ghazali Talbi	Professeur LIFL	**Président**
	Marc Pouzet	Professeur LRI	**Rapporteur**
	Olivier Sentieys	Professeur IRISA	**Rapporteur**
	Zbigniew Chamski	Philips Research	**Examinateur**
	Pierre Boulet	Professeur LIFL	**Directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Remerciements

Les remerciements de thèse cachent parfois, sous le couvert de formules de politesses convenues, l'indifférence ou le mépris réciproque qui s'est installé entre un thésard et son directeur de thèse. Mais ici, je veux exprimer ma sincère reconnaissance et ma profonde gratitude à mon directeur de thèse Pierre Boulet. En effet, Pierre Boulet a toujours été là pour écouter et comprendre les cheminements tortueux de ma pensée, il a toujours su me remettre au travail lorsque les sirènes de l'enseignement se faisaient trop fortes et surtout il m'a toujours témoigné un soutien indéfectible qui a permis cette délivrance miraculeuse : la soutenance.

Je tiens à remercier M. El-Ghazali Talbi d'avoir accepté de présider mon jury de thèse. Je remercie également M. Marc Pouzet et M. Olivier Sentieys d'avoir rapporté cette thèse dans des délais aussi brefs ; j'ai été particulièrement sensible au soin avec lequel ils ont relu ce mémoire et ceci malgré le peu de temps dont ils disposaient. Enfin, je remercie M. Zbigniew Chamski d'avoir fait partie de mon jury.

C'est au sein de l'équipe WEST que s'est déroulée cette thèse et je remercie donc son responsable Jean-Luc Dekeyser de m'avoir accueilli. L'évolution de l'équipe au cours de ces dernières années ne me permet pas d'en citer tous ses membres. De Chadi à Sébastien, en passant par les deux charognes « warcraftiennes » Stéphane et Christophe, l'inénarrable Alexandre ou le jovial Ashish, la liste est bien trop longue pour être énumérée, mais je les assure tous ici de ma profonde sympathie.

Deux membres de l'équipe WEST m'ont toutefois plus particulièrement influencé : Philippe Marquet et Julien Soula. Philippe Marquet a été le maître d'œuvre de mes enseignements au FIL. J'ai été et je suis toujours admiratif devant l'exigence, la qualité et la pédagogie des cours qu'il dispense ; il reste pour moi l'exemple à suivre. Julien Soula, lui, n'a jamais cessé de répondre à mes innombrables questions sur ARRAY-OL en faisant preuve de son éternelle bonne humeur, mais il m'a surtout fait profiter de ses inépuisables connaissances sur le monde Unix en me guidant tous les jours plus avant dans les abîmes de la « geekitude ».

Je ne peux évoquer ces longues années de thèse passées au LIFL sans saluer ici tous ceux que j'ai pu rencontrer et notamment : Alexandre Courbot, Julien Derveeuw, Vincent Houseaux, Nicolas Jozefowiez, Anne-Françoise Le Meur, Jean-Luc Levaire, Alexis Muller, Alexandre Sedoglavic, Léopold Weinberg et naturellement le *Professeur* David Simplot.

Au delà des murs du LIFL et du carcan dans lequel s'enferme tout thésard, il existe une

vie dans laquelle j'ai pu compter sur mes amis malgré mon lointain exil lillois : Hubert, David, la merveilleuse Charlotte, Marion, Marjorie, François, Cécile, Marie, Bertrand. Leur amitié a été le meilleur des réconforts lorsque la fatigue et la lassitude rendaient toujours plus improbable une soutenance qui ne cessait de s'éloigner.

Au delà des mots, il y a ma famille.

Table des matières

Introduction

Ptah., $L_2$1, 14 [nº 53]

La vie a du relief ! Certes, ceci n'est pas une nouveauté, mais derrière cette banalité se cache le fait que nous vivons bel et bien dans un monde à plusieurs dimensions. Or rien ne déroge à cette règle, pas même les données manipulées par des applications de traitement du signal. Le traitement du signal est une discipline qui consiste à analyser et interpréter des signaux qui peuvent provenir de sources très diverses, mais la plupart sont des signaux électriques ou devenus électriques à l'aide de capteurs. On peut donc facilement imaginer des signaux tels que les images de télévision qui comportent naturellement plusieurs dimensions. Pourtant il n'existe que très peu de modèles d'applications qui soient capables de prendre en compte cet aspect multidimensionnel.

Nous nous sommes donc intéressés à ce problème, mais en nous limitant à un domaine particulier du traitement du signal : le traitement du signal systématique (TSS). Ce domaine correspond à la première phase de traitement des signaux et consiste en l'application de traitements très réguliers, indépendants des données. L'intérêt du TSS provient justement de la régularité de ses traitements qui laissent espérer de grandes possibilités d'optimisation.

Pour manipuler des applications de TSS, nous utilisons le langage ARRAY-OL qui a été inventé chez Thales Underwater Systems (TUS) dans le but exclusif de modéliser ce type d'applications. ARRAY-OL est un modèle de description qui a la faculté d'exprimer les dépendances de données au sein d'une application. En revanche, ARRAY-OL ne fournit aucune méthodologie pour l'exécution de ces applications. Il faut donc projeter ARRAY-OL sur un modèle de calcul pour exécuter les applications. Entre ces deux phases de modélisation et d'exécution, il est possible de réaliser une phase d'optimisation qui tienne compte des particularités d'ARRAY-OL pour pouvoir exécuter « intelligemment » les applications.

Le but de cette thèse est d'étudier les problèmes liés à ces optimisations. Nous nous basons pour cela sur les travaux qui ont été préalablement effectués sur ce sujet : il s'agit notamment de la thèse de Julien Soula qui décrit les principes de base de ces optimisations. Nous reprenons les résultats qui y sont présentés afin de fournir un ensemble complet d'opérations appelées « transformations » qui sont destinées à appliquer ces optimisations. Nous verrons également différentes méthodes d'exécution pouvant être appliquées à une application ARRAY-OL, ce qui nous permettra de mettre en lumière l'impact des optimisations.

Plan du manuscrit

Ce manuscrit aborde les différents objectifs que nous venons d'évoquer en essayant de répondre à quatre grandes questions. Comment modéliser une application ? Comment réaliser une optimisation ? Quelles sont ces optimisations ? Comment exécuter une application ?

Chapitre 1 : Spécification multidimensionnelle

Dans le premier chapitre, nous traitons de la problématique de modélisation des applications de traitement du signal systématique. Nous étudions pour cela plusieurs modèles basés sur les flots de données synchrones comme LUSTRE ou SDF. Puis, nous introduisons le modèle ARRAY-OL.

Chapitre 2 : Optimisation des applications ARRAY-OL

Ce chapitre est consacré à l'étude de l'optimisation des applications décrites en ARRAY-OL. Nous présentons différentes méthodes et formalismes pouvant permettre ces optimisations, nous introduisons finalement le formalisme ODT qui a été spécialement conçu pour ARRAY-OL.

Chapitre 3 : Transformation des applications ARRAY-OL

Dans ce chapitre, nous utilisons le formalisme ODT décrit au chapitre précédent pour constituer une « boîte à outils » de transformations capables d'effectuer de simples modifications ou des optimisations complexes sur des applications ARRAY-OL.

Chapitre 4 :Exécution d'ARRAY-OL

À partir des résultats présentés dans le chapitre 3, nous analysons la projection d'ARRAY-OL sur différents modèles de calculs et nous montrons l'importance de ces résultats pour une projection « intelligente ».

Conclusion

En conclusion, nous présentons le bilan des travaux effectués et le détail des contributions apportées, puis nous énonçons quelques perspectives envisageables.

Chapitre 1

Spécification multidimensionnelle

Ἓν οἶδα ὅτι οὐδὲν οἶδα.

Platon, *Apologie de Socrate*, 21 a.

1.1 Traitement du signal systématique et flots de données

1.1.1 Le traitement du signal intensif

La partie du traitement de signal (TS) qui nous intéresse est sa partie la plus intensive, composée du traitement de signal systématique (TSS) et du traitement de données intensif (TDI) plus irrégulier. Le TSS correspond à la première phase de traitement des signaux et consiste principalement à l'application de filtres et à des traitements très réguliers (indépendants de la valeur des signaux) appliqués systématiquement aux signaux d'entrée pour en extraire les caractéristiques intéressantes. Celles-ci sont ensuite traitées par des calculs plus irréguliers (dépendants de la valeur de ces grandeurs) dans la phase de TDI.

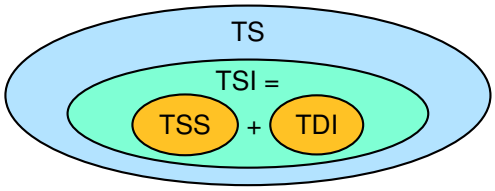


FIG. 1.1: Schéma du TS

Exemples d'applications : Ce schéma en deux phases se retrouve dans beaucoup d'applications de traitement de signal ou de l'image. En voici quelques exemples représentatifs venant des collaborations de l'équipe WEST avec des partenaires industriels.

– **Récepteur de radio numérique :** cette application en émergence fait appel à une partie frontale de TSS consistant à la numérisation de la bande de réception, la sélection du canal et l'application de filtres permettant d'éviter les parasites. Les données fournies par ces traitements systématiques sont ensuite envoyées dans le décodeur dont le traitement est plus irrégulier (synchronisation, démodulation, etc).

- **Traitement sonar :** une chaîne de traitement sonar classique se compose d'une première étape systématique : la veille bande large suivie d'un traitement de données : la poursuite. La première phase prend en entrée les signaux produits par les hydrophones (microphones répartis autour du sous-marin) et, par une suite de traitements systématiques produit des voies, couples (direction, intensité), représentant les échos captés. Ces échos sont ensuite analysés par la poursuite pour identifier et suivre au cours du temps les objets les produisant.
- **Encodeur/décodeur JPEG-2000 :** JPEG-2000 est un nouveau format standard de compression d'images. Le fonctionnement de l'encodeur **[**1**]** suit le même schéma en deux phases. La première partie (du prétraitement à la décomposition en ondelettes) est systématique. C'est dans la deuxième partie de l'encodage qu'apparaissent des traitements irréguliers (quantification, deux étages d'encodage). Le décodeur fonctionne exactement à l'inverse de l'encodeur et fait donc se suivre une phase de TDI et une phase de TSS.
- **Convertisseur** $16/9$ **-** $4/3$ **:** La convertion d'un signal télévisé au format $16/9$ vers un signal au format $4/3$ se déroule en deux étapes **[**44**]** : le signal est d'abord interpolé afin de rajouter artificiellement des lignes sur chaque image, puis une partie seulement de ces lignes est conservée pour former le signal de sortie au format $4/3$. L'intégralité des traitements est indépendante de la valeur du signal, il s'agit donc de TSS.

Bien que très diverses, toutes ces applications font donc parties du traitement du signal intensif. Elles réalisent toutes des traitements réguliers sur les données et certaines effectuent en plus des traitements irréguliers. Mais trois d'entre elles partagent également un autre point commun : la manipulation de structures de données multidimensionnelles.

- **Traitement sonar :** le signal d'entrée est constitué par les données recueillies par les hydrophones autour d'un sous marin, l'utilisation d'une FFT comme premier traitement ajoute une dimension, le signal est alors bi-dimensionnel ;
- **Encodeur/décodeur JPEG-2000 :** les images sont traitées comme étant bi-dimensionnelles, leurs dimensions ne sont pas linéarisées ;
- **Convertisseur** $16/9$ **-** $4/3$ **:** le signal est bi-dimensionnel, il est constitué de tous les points des 625 lignes des images de télévision.

À ces multiples dimensions, il convient biensûr d'ajouter la dimension temporelle.

Caractéristiques de la partie TSS : Toutes les applications que nous venons de décrire comportent une phase de TSS. Or la regularité des traitement du TSS offrent plusieurs caractéristiques intéressantes :

- les valeurs traités par les calculs sont ordonnées suivant des grandeurs physiques et se structurent en tableaux ;
- au fil des calculs la taille de ces tableaux peut être amenée à changer, leur nombre de dimensions peut également évoluer ;
- les tableaux peuvent avoir des dimensions cycliques ou de taille infinie (pour représenter le temps par exemple) ;
- les calculs effectués sur les données sont indépendants de la valeur de ces données ;
- les calculs ne sont généralement pas très différents d'une application à l'autre et sont majoritaiments composés de produits scalaires, de transformées de Fourier (FFT), etc. ;
- les calculs prennent toujours des ensembles de données de formes identiques et de même cardinalité, le traitement des données est régulier.

L'énumération de ces caractéristiques montre que la nature des calculs importe peu, seule l'interaction de ces calculs avec les données est importante. Une modélisation capable d'exprimer ces différentes caractéristiques permettrait d'effectuer alors de nombreuses optimisations.

1.1.2 Le modèle flots de données

Présentation : Le modèle flots de données peut être abordé sous deux axes de lectures différents. Dans un premier temps, il peut être étudié à travers des langages « graphiques » comme SDF et KPN. Dans un langage « graphique », une application est modélisée sous la forme d'un graphe où les nœuds représentent les traitements à effectuer et où les arcs représentent les flots de données. Mais il peut également être vu comme le support de langages « textuels » comme Signal, LUSTRE et Lucid Synchrone. Les langages « textuels » sont des langages de programmation dans lesquels les variables sont définis par la suite des valeurs qu'elles prennent au cours de l'exécution, ces valeurs sont fixés à l'aide d'équations.

Il existe une autre dualité dans l'analyse du modèle flot de données. En effet, on peut considérer que les langages « graphiques » et « textuels » reposent sur une sémantique différente : dans le premier cas les données qui sont produites existent jusqu'à ce qu'elles soient consommées, dans le deuxième cas les données ne sont produites et consommées qu'à certains tops d'horloge qui marquent leur unique moment d'existence.

Cependant malgré ces différences, ces deux types de langages permettent de modéliser des applications aux propriétés proches de celles qui nous intéressent en TSS. William Thies, Michal Karczmarek et Saman Amarasinghe établissent dans **[**60**]** une liste de ces propriétés :

1. **D'importants flots de données :** la première caractéristique d'une application basée sur les flots de données est la gestion de séquences de données de grandes tailles voire de taille infinie. Les données sont généralement produites par une source, puis servent à affecter différents calculs avant d'être écartées.
2. **Indépendance des filtres :** une application peut être vue comme une suite de transformations appliquées sur le flot de données. On appelle filtres les unités de calcul appliquant ces transformations. À chaque exécution, ces unités lisent des données sur le flot entrant avant de s'en servir pour calculer un ou plusieurs résultats qu'ils placeront sur le flot sortant. Ces filtres sont indépendants et peuvent être exécutés de manière autonome. Une application peut donc être vue comme un graphe de filtres reliés par les flots de données.
3. **Une exécution « stable » :** l'exécution d'une application doit être la plus régulière possible, elle doit respecter un schéma d'exécution. Ainsi les filtres sont généralement exécutés dans le même ordre.
4. **D'éventuelles modification de la structure du flux :** bien que les calculs suivent un schéma d'exécution « stable », il se peut que ce schéma soit amené à être modifié au cours de l'exécution pour faire face à une situation donnée. Toutefois l'ensemble des schémas possibles doit être connu lors de la phase de compilation.
5. **D'éventuelles communications en dehors du flux :** les filtres peuvent être amenés à communiquer occasionellement en dehors du flux, il s'agit généralement de petites informations de contrôle.
6. **Des calculs « hautes performances » :** le plus souvent les applications sont soumises à des contraintes de temps réel, mais pour les applications destinées à l'embarqué, il faut

ajouter des contraintes de taille du code, d'occupation de la mémoire et de consommation

Restrictions : À la vue de ces propriétés, on peut déduire que le modèle flot de données est adapté à la modélisation de nos applications. En effet, les trois premières propriétés correspondent aux caractéristiques principales du TSS. On note également que les trois dernières propriétés ne nous concernent pas directement ; il n'y a pas besoin d'exprimer d'éventuelles modification de la structure du flux ou d'eventuelles communications en dehors du flux pour modéliser la phase de TSS d'une application : le TSS ne comporte aucun flot de contrôle, il est uniquement limité aux flots de données. De plus, nous n'exprimerons pas les diverses contraintes qui se rattachent aux calculs « hautes performances », mais nous verrons, dans la suite de ce document, que les optimisations que nous proposons essayent d'y repondre.

Nous appliquons une autre restriction sur notre analyse du modèle à flots de données en ne considérant que les modèles synchrones. Le terme synchrone n'a pas le même sens selon qu'il est utilisé dans le cadre des langages « textuels » ou dans celui des langages « graphiques ». Dans le premier cas, on appèle synchrone un modèle où les temps de communications et de traitement des données ne sont pas pris en compte, dans le deuxième cas le terme synchrone implique que les nœuds du graphe sont obligés de consommer et de produire un nombre de données entier qui a été fixé dès la conception de l'application. Or nous avons vu que nos applications répondent déja à ces deux exigences, nous pouvons donc limiter notre étude aux langages à flot de données synchrones qu'ils soient « textuels » ou « graphiques ».

Conclusion : Nous avons donc le choix entre deux types de langages pour modéliser nos applications : les langages « textuels » et les langages « graphiques ». Ces derniers comportent plusieurs aspects intéressants. Ainsi, une modélisation de haut niveau sous la forme d'un graphe permet de s'éloigner des problèmes liés aux traitements des données pour se concentrer sur l'aspect gestion du flux et expression des dépendances. En TSS, les valeurs des données n'influent pas sur les calculs, il semble donc naturel de vouloir dissocier l'aspect traitement des données de la gestion du flux et des dépendances. De plus, la représentation d'une application sous la forme d'un graphe est simple et intuitive, notre but n'est pas de programmer une application mais de la modéliser. Ces différentes raisons nous ont conduit à nous intéresser aux langages « graphiques ».

Toutefois, nous allons étudier au préalable différents langages « textuels » dans le but de comprendre comment ils définissent leur intéraction avec le flot de données. Cette interaction est primordiale puisqu'elle constitue le principal point de comparaison entre les langages « textuels » et « graphiques ».

1.1.3 Flots de données synchrones

Nous abordons dans cette section différents langages à flot de données synchrones et à representation « textuelle ». La plupart de ces langages reposent sur la programmation fonctionnelle, nous étudierons cependant un cas particulier basé sur la programmation impérative et appelé StreamIt.

1.1.3.1 L'approche fonctionnelle

Présentation de LUSTRE : LUSTRE [31,55] est un langage à flot de données synchrone créé par N. Hallbwachs, P. Capsi, P. Raymond, et D. Pilaud, il est utilisé pour la programmation des systèmes réactifs synchrone. Un système réactif synchrone intéragit de façon permanente avec son environnement et il doit répondre en temps réel aux stimuli que lui envoit son environnement. On retrouve les systèmes réactifs dans des domaines aussi variés que l'automobile, l'aviation, le nucléaire, les appareils médicaux... LUSTRE est également utilisé comme langage « textuel » de Scade. Scade est un logiciel développé par Esterel Technologies qui propose un langage graphique ayant une équivalence sémantique avec LUSTRE et qui fournit un ensemble d'outils pour la simulation, le calcul de preuve et la génération de code.

Fonctionnement de LUSTRE : LUSTRE est un langage fonctionnel qui manipule des flux. Une fonction dans lustre est appelée nœud *(node)*, elle peut prendre plusieurs paramètres d'entrée et de sortie. Les paramètres de sortie sont définis comme des fonctions des paramètres d'entrée. Tous ces paramètres sont des flux. L'originalité de LUSTRE réside dans la composition de ces flux. En effet en LUSTRE, un flot de données est composé à la fois d'une suite de valeurs et d'une horloge.

Les horloges sont à la base du fonctionnement de LUSTRE. Une horloge est une suite d'instants (t_0, t_1, etc) qui constitue une discrétisation du temps. Il existe une horloge de référence qui est valable pour tout le système et pour laquelle chaque instant t_i marque l'exécution d'un cycle. Mais chaque flot est également doté de sa propre horloge. Une horloge de flot est obligatoirement une sous horloge de l'horloge de référence, chaque top de cette horloge marque le moment où le flot est défini (*cf* tableau 1.2). Une sous horloge peut être vue comme un flot de booléens de valeur *true* pour chaque top. Les horloges permettent ainsi d'utiliser des nœuds ayant un rythme de fonctionnement différent.

horloge de référence	t_0	t_1	t_2	t_3	t_4	t_5	t_6
valeur du flot	0	1	2	3	4	5	6
sous horloge			t_0		t_1	t_2	
valeur du flot			0		1	2	

FIG. 1.2: Principe des horloges en LUSTRE.

LUSTRE utilise un ensemble d'opérateurs sur les variables pour manipuler ces flux. Une variable X est définie à partir d'une expression E ; une expression est composée de variables, de constantes et d'opérateurs qu'ils soient arithmétiques, booléens, relationnels ou conditionnels. Une variable est définie comme étant une suite de valeurs, ces opérateurs agissent donc sur l'ensemble du flux et non sur une valeur particulière. Mais LUSTRE propose un autre type d'opérateurs : les opérateurs temporels, il en existe quatre différents :

1. « pre » qui renvoie la valeur du flux à l'instant précédent ;
2. l'opérateur « $\rightarrow$ » qui permet de modifier la valeur d'un flot de données à l'instant initial, t_0 ;
3. « when » qui permet de sélectionner des valeurs d'un flux à partir d'un flux booléens de même horloge ;

4. « current » qui permet d'accorder un flux sur une horloge plus rapide en prenant lorsque le flux n'est pas défini la dernière valeur pour laquelle il l'était.

Un exemple d'utilisation de ces opérateurs est donnée dans les tableaux 1.3 et 1.4. En Scade, tous ces opérateurs constituent les nœuds d'un graphe servant à représenter le traitement effectué par une fonction.

horloge de référence	t_0	t_1	t_2	t_3	t_4	t_5
x	5	8	2	3	13	5
pre x	null	5	8	2	3	13
$9 \rightarrow x$	9	8	2	3	13	5

FIG. 1.3: Les opérateurs « pre » et « $\rightarrow$ » en LUSTRE.

horloge de référence	t_0	t_1	t_2	t_3	t_4	t_5
B	false	true	false	true	true	false
X	x_1	x_2	x_3	x_4	x_5	x_6
$Y = X$ when B		x_2		x_4	x_5	
$Z =$ current Y	null	x_2	x_2	x_4	x_5	x_5

FIG. 1.4: Les opérateurs « when » et « current » en LUSTRE.

LUSTRE et Scade présentent donc le double intérêt de la definition graphique de l'application et de l'équivalence avec un langage « textuel ».

Les tableaux dans LUSTRE : Les tableaux ont été introduits dans LUSTRE pour aider à la description de circuits réguliers **[**56**]** ; la compilaton de ces tableaux a été ensuite optimisée par Lionel Morel **[**48**]**. On définit un flux de tableaux en désignant le type des éléments, t, et en donnant le nombre de ces élements, n, (ce nombre doit être connu à la compilation). Ainsi, $int\hat{}m\hat{}n$ désigne une matrice d'entiers à m lignes et n colonnes. Les manipulations élémentaires sur les tableaux sont possibles : $A[i..j]$ désigne tous les élements du tableau entre i et j ; $A|B$ effectue une concaténation des tableaux A et B. De plus, tous les opérateurs du langage LUSTRE tel que « pre » ou « $\rightarrow$ » ont été étendus pour supporter les tableaux.

Morel propose en outre l'introduction d'itérateurs de tableaux pour générer des boucles lors de la compilation. Ces itérateurs sont au nombre de quatre :

1. *map* : consiste à appliquer la même fonction à tous les éléments d'un ou plusieurs tableaux ;
2. *red* : permet de calculer un accumulateur en parcourant un ou plusieurs tableaux ;
3. *fill* : permet de remplir un ou plusieurs tableaux en effectuant des itérations sur une fonction à partir d'une valeur initiale ;
4. *map_red* : est un itérateur générique capable de générer les autres ;

LUSTRE est donc en mesure de fournir un support complet des tableaux multidimensionnels. Cependant cette utilisation est restreinte aux simples nœuds, il n'est pas possible d'exprimer

explicitement les dépendances entre les nœuds. Cette information doit être extraite des manipulations effectués dans les nœuds sur ces tableaux, ce qui parait très difficile. Or nous verrons que dans un langage de flux dédié au multidimensionnel l'expression des dépendances entre les nœuds joue un rôle crucial.

Signal et Lucid Synchrone : Il n'est pas possible de parler des langages à flots de données synchrone sans évoquer Signal et Lucid Synchrone. Lucid Synchrone **[**14**]** est un langage fonctionnel qui repose sur l'utilisation dans un langage synchrone des typages forts que l'on rencontre dans la famille des langages ML. Signal **[**7**]** diffère de LUSTRE par l'absence éventuelle d'horloge de référence pour cadencer l'ensemble des objets d'un programme. Les différentes horloges peuvent donc être indépendantes. Le calcul d'horloges de Signal sait déterminer automatiquement s'il existe une horloge de référence contenant les multiples horloges initiales du programme. Il la synthetise lorsque celle-ci existe et l'on obtient ainsi un nouveau programme Signal équivalent à un programme Lustre.

Cependant la gestion du flot données, des dépendances ou des tableaux n'étant pas fondamentalement différentes de celle de LUSTRE, nous ne développerons pas plus l'étude de ces langages.

1.1.3.2 L'approche impérative : StreamIt

StreamIt **[**60, 30**]** est une exception dans les langages « textuels » à flots de données : il n'est pas fonctionnel contrairement aux autres langages que nous venons de voir. StreamIt, pour *stream-MIT*, est un langage impératif utilisant la programmation orienté objet, il est actuellement en cours de développement au MIT. Il a été créé dans le but de permettre une optimisation maximum tout en fournissant une abstraction de haut au niveau pour la programmation des flots de données synchrones. StreamIt est ainsi capable d'exprimer toutes les propriétés dont nous avons repris la description dans la section 1.1.2 page 5. Pour réaliser les optimisations espérées, un compilateur a également vu le jour.

Un programme StreamIt reprend la syntaxe du langage JAVA, il est d'ailleurs possible d'utiliser un compilateur JAVA sur un code en StreamIt après une phase de traduction. En StreamIt les filtres constituent la classe de base, ces classes sont constituées principalement de deux méthodes :

- la méthode `work` est la plus importante car elle contient la liste des traitements réalisés par le filtre. En outre, c'est dans cette méthode que le filtre peut communiquer avec les autres filtres de l'application. Il utilise pour celà des FIFOs qui sont traités comme des attributs de la classe `filter`. L'intéraction avec ces FIFOs se fait à l'aide de trois méthodes `push`, `pop` et `peek` qui sont semblables aux méthodes habituelles de manipulations de FIFOs ;
- la méthode `init` est utilisée pour initialiser le filtre et les FIFOs. Il est ainsi possible de positionner des valeurs initiales dans les FIFOs avant le début de l'exécution. Mais surtout la méthode permet de positionner le nombre de données qui seront consommées ou produites par `push`, `pop` et `peek` dans la méthode `work`.

Il est ensuite possible de relier ces filtres en utilisant trois méthodes différentes qui représentent chacune une topologie différente : un lien direct, une boucle et une dissociation suivie d'un regroupement. En outre, StreamIt propose également d'utiliser un système de messages pour transmettre des informations de contrôle, mais nous n'aborderons pas cette partie puisqu'elle ne nous intéresse pas directement.

L'interaction de StreamIt avec le flot de données est donc relativement basique. On peut considérer que StreamIt est le pendant « textuel » du langage « graphique » SDF que nous allons étudier dans la section 1.2 et ceci bien que StreamIt dispose de quelques fonctionnalités supplémentaires.

1.1.4 Flots de données multidimensionnels

Nous présentons ici un langage à flots de données, qui n'est certes pas synchrone, mais qui permet de gérer des flots de tableaux multidimensionnels de taille quelconque et surtout de forme quelconque.

ALPHA [46] est développé au laboratoire Irisa de Rennes, il est généralement associé à l'environnement MMALPHA qui est une interface basée sur Mathematica pour la manipulation du langage ALPHA. L'intérêt d'ALPHA est de fournir un langage de haut niveau pour synthétiser des architectures VLSI[1]. En effet, ALPHA est un langage fonctionnel, à assignation unique et fortement typé qui permet de représenter les algorithmes sous forme d'équations récurrentes [35].

ALPHA ne présente pas de particularismes notables concernant la déclaration des fonctions, en revanche les variables sont définies à l'aide de fonctions sur le domaine $\mathbb{Z}^n$, En effet, les données manipulées par ALPHA sont multidimensionnelles : elles correspondent à des unions de polyèdres convexes. Leurs formes ne sont donc pas restreintes à de simples tableaux rectangulaires. L'exemple suivant montre la déclaration d'une variable dont le domaine est l'ensemble des points dans le triangle : $0 \leq i \leq j; j \leq 10$:

```
a : {i,j | 0 ≤ i ≤ j; j ≤ 10}
```

Afin d'accéder aux différentes valeurs de ces données, il est possible de faire des restrictions sur les domaines. On se sert pour cela de l'instruction `case`.

```
 a =
case
{i,j | j = 0 } : 0.(i,j->);
{i,j | j > 0 } : a.(i,j->i,j-1)+1.(i,j->);
esac;
```

Dans cet exemple, nous avons $a[i,j] = 0$ lorsque $j = 0$ et $a[i,j] = a[i,j-1] + 1$.

ALPHA se révèle donc être en mesure d'exprimer de façon simple des formes de données très complexes, mais il s'avère incapable de gérer les accès cycliques dont nous avons besoin (*cf* section 1.1.1 page 4). En outre, nous n'avons pas besoin de gérer des formes de données aussi complexes et nous pouvons donc nous contenter de simples tableaux à plusieurs dimensions.

1.1.5 Conclusion

Nous venons d'étudier plusieurs langages à flots de données synchrones qui présentent tous des particularismes intéressants, toutefois aucun d'entre eux ne répond exactement à nos désidérata. Nous introduisons ci-dessous deux langages à flots de données conçus spécialement pour la gestion de flots de données multidimensionnels.

[1] VLSI : Very Large Scale Integration caractérise les circuits intégrés de très haute intégration.

1.2 SDF et ses extensions

Cette section fait l'objet de l'étude du langage SDF [41,42] et de ses dérivés MDSDF [39,15] et GMDSDF [50,49,51]. SDF (Synchronous DataFlow) est un langage à flot de données synchrones. Généralement, les flots de données sont utilisés pour décrire des applications du traitement du signal par des graphes, en représentant les fonctions par des nœuds et les données par les arêtes du graphe. L'ajout du terme synchrone implique que le nombre de données consommées et produites soit connu dès la conception de l'application, ce qui permet de réaliser des ordonnancements statiques. Réservé à des applications mono-dimensionnelles, SDF a été modifié pour le support des applications multidimensionnelles.

1.2.1 SDF : Synchronous DataFlow

Le modèle SDF a été créé et développé par Edward A. Lee en 1986 [41,42]. Lee l'a ensuite intégré à PTOLEMY son environnement de modélisation et de simulation d'applications pour l'embarqué.

En SDF, une application est donc décrite par un graphe orienté acyclique dont chaque nœud consomme et produit des données respectivement sur ses arêtes entrantes et sortantes. Dans PTOLEMY, ces données sont appelées jetons, *tokens*, et les nœuds sont appelés acteurs, *actors*. La figure 1.5 montre le graphe d'une application décrite en SDF.

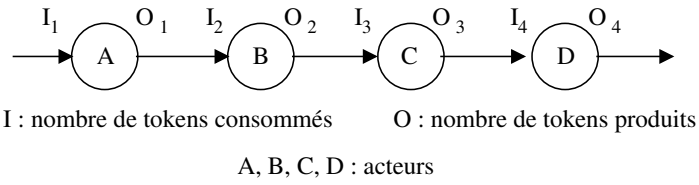


FIG. 1.5: Une application SDF

1.2.1.1 Caractéristiques d'une application SDF

Les caractéristiques principales sont :

- Le nombre de données consommées et produites par un nœud à chaque exécution est fixé dès la conception de l'application. Ce nombre doit être connu lors de la modélisation. En revanche aucune information n'est nécessaire sur les traitements effectués par ces acteurs.
- La valeur des jetons ne doit en rien modifier le flot de données. Les données ne doivent servir qu'aux calculs réalisés par les acteurs et ne doivent pas changer le comportement de l'application. Le flot de contrôle est donc indépendant des données.

Grâce à ces caractéristiques, l'application est définie statiquement. Il est donc possible d'utiliser les propriétés formelles qui en découlent afin :

- de détecter les interblocages dès la conception ;
- d'ordonnancer l'application dès la modélisation ;
- d'avoir une exécution déterministe (l'exécution se déroule toujours de la même façon) ;
- d'avoir une exécution dans un temps fini et avec une consommation de mémoire finie.

À l'aide de ces propriétés, SDF permet de tester la validité d'une application dès sa phase de conception, ce qui est bien entendu une grande force.

1.2.1.2 Calcul de l'ordonnancement

Le calcul de l'ordonnancement se base sur la constatation suivante : lorsque l'application est éxécutée, le nombre total de jetons produits par une tâche est égal au nombre de jetons consommés par la tâche suivante. Soit I_x et O_x le nombre de jetons respectivement consommé et produit par la x^e tâche et soit r_x le nombre d'exécution de cette tâche (*cf* figure 1.5), on peut alors écrire l'équation suivante :

$$r_x O_x = r_{x+1} I_{x+1} \tag{1.1}$$

On obtient ainsi un système d'équations ; ce système n'a soit aucune solution, soit une infinité. Dans le premier cas, le graphe est inconsistant et il ne peut être ordonnancé. Dans le deuxième cas, toutes les solutions sont multiples d'une seule et même solution ($\overrightarrow{r}$). L'ensemble des solutions est donc décrit par $k\overrightarrow{r}$ avec $k \in \mathbb{N}^+$.

Lorsque $\overrightarrow{r}$ a été calculé, il est possible, grâce aux résultats présentés dans **[**41**]**, de construire un graphe de dépendance. Le calcul de $\overrightarrow{r}$ et la construction du graphe de dépendance étant automatiques, il n'y a donc pas de difficultés pour obtenir l'ordonnancement.

Exemple Soit l'exemple présenté sur la figure 1.6 : la première tâche produit deux jetons et la deuxième en consomme trois. La résolution du système d'équation donne $\overrightarrow{r} = \binom{3}{2}$ et le graphe de dépendance obtenu est présenté sur la droite de la figure.

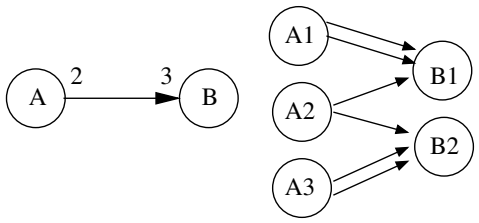


FIG. 1.6: Graphe de dépendance d'une application SDF

1.2.1.3 Utilisation des états et des délais

Une autre opportunité offerte par SDF est l'utilisation des « délais ». Un délai est un ensemble de jetons qu'on peut placer sur une arête. Lors de l'exécution de l'application, la tâche, se trouvant en amont de l'arête, produira des jetons, mais ceux-ci ne seront consommés par la tâche en aval qu'après les jetons du délai. Les délais permettent donc de prépositionner des valeurs initiales sur les arêtes.

Un « état » est un délai particulier qui autorise une tâche à utiliser le résultat d'une itération lors de l'itération suivante. Ainsi les n-premiers jetons de l'état sont utilisés par les n-premières itérations de la tâche, puis celle-ci utilise les résultats qu'elle a produits. L'utilisation des états est extrêmement utile : un exemple simple est la somme d'un vecteur où l'état ne comporte qu'un seul jeton de valeur 0.

L'utilisation des états et des délais ne modifie pas la manière de calculer un ordonnancement.

1.2.2 MDSDF : MultiDimensional Synchronous Dataflow

SDF autorise la modélisation de flots de données mono-dimensionnelles en spécifiant les dépendances entre les tâches. Les jetons transportés par ces flots de données sont de nature quelconque, il peut notamment s'agir de vecteurs ou de tableaux. Le principe de MDSDF [39, 15] n'est donc pas de transporter des flots de données multi-dimensionnelles, puisque cela est déjà possible en SDF, mais d'exprimer les dépendances entre les tâches.

Le fonctionnement de MDSDF est similaire à celui de SDF. Ainsi, il suffit juste de préciser pour une tâche le nombre de données consommées et produites sur chacune des dimensions du flot. Ce principe est illustré par la figure 1.7. Le flot représenté est de forme bidimensionnelle, la première tâche A produit un rectangle de jetons de taille $O_{A,1}$ sur la première dimension et de taille $O_{A,2}$ sur la deuxième pour chaque itération.

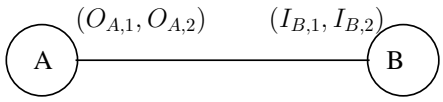


FIG. 1.7: Une application MDSDF

1.2.2.1 De l'intérêt de MDSDF

L'utilisation de MDSDF permet d'exprimer de manière plus exacte une application. Les figures 1.8 et 1.9 montrent la même application décrite respectivement avec MDSDF et avec SDF. En MDSDF (figure 1.8), il est évident que la première tâche produit un tableau de taille 40×48 qui est consommé par morceaux de 8×8 par la deuxième tâche. En SDF il apparaît simplement que la première tâche produit trente jetons et que la deuxième les consomme un par un. SDF ne permet donc pas d'exprimer les dépendances de façon optimale lorsque la structure du flot de données est multi-dimensionnelle.

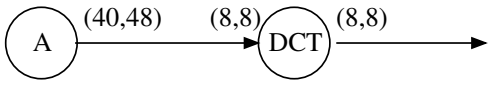


FIG. 1.8: Une application simple en MDSDF.

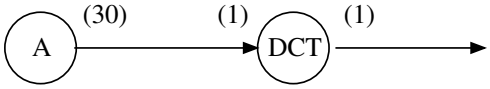


FIG. 1.9: La même application en SDF.

Mais surtout MDSDF rend possible la modélisation d'application qui ne pouvait être décrite en SDF et ceci même avec une linéarisation des dimensions. La raison en est simple, il n'est pas possible en SDF de spécifier le mode de construction du graphe de dépendances. La figure 1.10 montre que la première tâche produit une colonne de deux jetons et que la deuxième tâche consomme une ligne de 3 jetons. Les deux jetons de la colonne ne seront donc pas consommés par la même itération de la deuxième tâche comme le montre le graphe de dépendances. Or la modélisation de cette application en SDF, vue sur la figure 1.6 ne respecte pas cette contrainte.

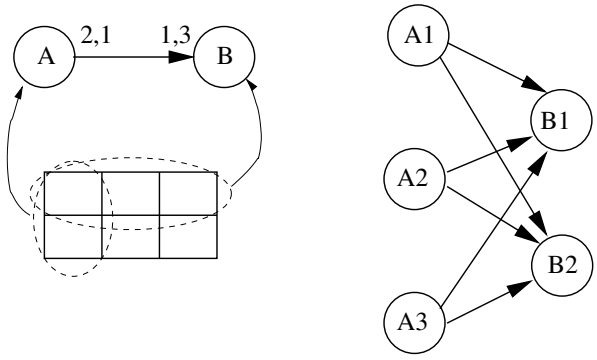


FIG. 1.10: Une application MDSDF qu'on ne peut transcrire en SDF

1.2.2.2 Calcul de l'ordonnancement

Le système d'équation de SDF est remplacé par plusieurs systèmes d'équations. En effet, un système est écrit pour chaque dimension :

$$\begin{aligned} r_{A,1}O_{A,1} &= r_{B,1}I_{B,1} \\ r_{A,2}O_{A,2} &= r_{B,2}I_{B,2} \end{aligned} \tag{1.2}$$

Chaque système est résolu de manière autonome. Les solutions obtenues sont de même forme que celles de SDF, chaque solution s'appliquant à une dimension. Le nombre total d'itérations d'une tâche est égal au produit du nombre d'itérations pour chacune des dimensions.

1.2.2.3 Particularité du modèle MDSDF

Dans les applications qu'on vient de voir en exemple, le nombre de dimensions du flot de données n'évolue jamais et il n'est d'ailleurs pas possible de créer ou de supprimer des dimensions. C'est pourquoi Lee propose des « acteurs clefs ». Ces derniers ne réalisent aucune opération, ils se contentent juste de modifier la structure du flot de données. D'ailleurs, il est important de rappeler que SDF ne manipule que les flots de données et jamais les données elles-mêmes. La liste des « acteurs clefs » n'est pas prédéfinie, Lee se contente d'en présenter quelques-uns. Le *downsample* sert par exemple à supprimer une dimension. La figure 1.11 représente un *downsample* qui consomme un tableau de $M \times N$ et qui produit un vecteur de taille M ; les éléments de la deuxième dimension ont été supprimés.

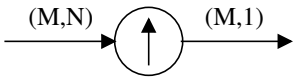


FIG. 1.11: Un *downsample*

À l'image de SDF, il est possible d'utiliser les « états » et les « délais » en MDSDF. Mais il s'agit désormais de n-uplet qui représente non pas des jetons initiaux mais des lignes et des colonnes de jetons initiaux. Les différences s'arrêtent là et l'utilisation se fait de manière identique à celle de SDF.

1.2.2.4 Conclusion

MDSDF rend possible l'utilisation de flot de données multidimensionnelles, mais certaines contraintes demeurent. Notamment la consommation et la production des données doivent être parallèles aux axes. Pour résoudre ce problème, Praveen K. Murthy and Edward A. Lee ont proposé une extension à MDSDF appelé GMDSDF.

1.2.3 GMDSDF : Generalized MultiDimensional Synchronous Dataflow

Le but de GMDSDF [50, 49, 51] est de fournir un formalisme capable de modéliser des applications avec une consommation ou une production des données non parallèles aux axes. En GMDSDF, les jetons ne sont plus produits en ligne et en colonne comme avec MDSDF, mais ils sont placés sur des treillis de points. Seuls certains points du treillis sont consommés ou produits par les tâches de l'application. Trois tâches spéciales sont introduites : la source, le « décimateur » et l'« expandeur », elles sont les seules à pouvoir manipuler les treillis.

1.2.3.1 Fonctionnement de GMDSDF

Nous introduisons ici les trois tâches spéciales de GMDSDF, puis les tâches ordinaires. Les principes de bases de GMDSDF restent cependant identiques à ceux de MDSDF.

La source : elle sert à créer et à définir la forme du treillis. Une source est toujours associée à une matrice exemple *(sampling matrix)* V. Cette matrice détermine la forme du treillis en définissant les points du plan[2] qui en font partie. La matrice treillis est constituée de l'ensemble des points $\overrightarrow{t} = V.\overrightarrow{n}, \forall \overrightarrow{n} \in \mathbb{N}^m, m \in \mathbb{N}$ Sur la partie gauche de la figure 1.12 , nous constatons que les points du treillis sont générés par la matrice exemple $\left(\begin{smallmatrix} 1 & -1 \\ 1 & 2 \end{smallmatrix}\right)$. De même on peut considérer que pour une application MDSDF bidimensionnelle $V = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

Une autre matrice est associée avec une source, il s'agit de la matrice support *(support matrix)* W. Elle est utilisée pour déterminer quels points du treillis seront produits. En effet une source ne produit pas tous les points du treillis. La matrice exemple ne sert qu'à donner la forme du treillis et la matrice support, elle, indique l'emplacement des éléments produits par la source. Le calcul de ces positions s'effectue de la manière suivante :

- à partir de la matrice support, il est possible de construire le « parallélépipède fondamental » *(fundamental parallelepiped)* en se basant à l'origine et en prenant les vecteurs de la matrice support comme les deux premiers cotés (*cf* partie droite de la figure 1.12) ;
- tous les points à coordonnées entières se trouvant à l'intérieur de ce parallélépipède sont appelés les « points renumérotés » *(renumbered points)* et sont notés $N(W)$; en outre, Lee montre que la cardinalité de cet ensemble est égale à la valeur absolue du déterminant de la matrice support $|N(W)| = |\det W|$;
- la multiplication des coordonnées des « points renumérotés » par la matrice exemple donne les coordonnées des points qui seront effectivement produits par la soure (*cf* figure 1.12).

Une source est définie par sa matrice exemple et sa matrice support.

[2] Il s'agit bien d'un plan, car comme nous le verrons plus tard, GMDSDF ne s'applique pas lorsque le nombre de dimensions est supérieur à deux.

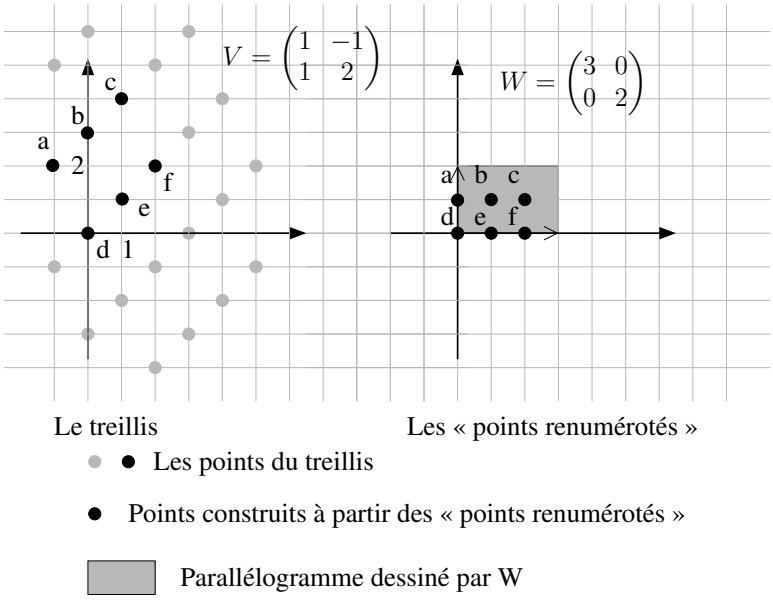


FIG. 1.12: Fonctionnement d'une source

Le « décimateur » et l'« expandeur » : Si la source permet de créer des treillis, le « décimateur » et l'« expandeur » permettent eux d'en modifier la forme : le premier en enlevant des points, le deuxième en en rajoutant. Ils sont respectivement définis par leur matrice de « décimation » (M) et d'« expansion » (L).

Lee et Murthy notent respectivement V_e et V_f les matrices exemples d'entrées et de sorties. De la même facon, il note W_e etW_f les matrices supports. Les liens unissant les treillis d'entrées de sorties sont données par les deux relations suivantes :

$$\begin{array}{lll} \text{« décimateur » :} & V_f = V_e.M & W_f = M^{-1}.W_e \\ \text{« expandeur » :} & V_f = V_e.L^{-1} & W_f = L.W_e \ . \end{array} \tag{1.3}$$

Ainsi, un « décimateur » de matrice $M = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$, utilisé sur un treillis de forme $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, ne garde qu'un point sur trois sur la dimension horizontale et un point sur deux sur la verticale. Un « expandeur » de matrice $L = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$ permet d'effectuer l'opération inverse.

Les autres tâches : Les tâches « ordinaires » consomment et produisent des points sur des treillis. Cependant elles n'utilisent plus un n-uplet comme en MDSDF mais une matrice support. Ainsi les points désignés ne sont plus nécessairement placés consécutivement sur une ligne ou sur une colonne. Mais ils peuvent être choisis de manière non parallèle aux axes et avec un décalage.

Les états et les délais : Les « états » et les « délais » sont également présents. Comme GMDSDF ignore les notions de lignes et de colonnes, un « état » ou un « délai » sont vus désormais comme un décalage de l'origine lors du calcul du « parallélépipède fondammental ». Le décalage des « points renumérotés » implique également un décalage des points qu'ils désignent. Les valeurs initiales des « états » et des « délais » servent donc à combler ce décalage.

1.2.3.2 Calcul de l'ordonnancement :

En MDSDF, il est possible de calculer simplement l'ordonnancement en résolvant le système d'équations. Ce système est simple à écrire car les dimensions sur lesquelles sont consommées et produites les données sont parallèles aux axes. Mais en GMDSDF ce n'est pas la cas et si une tâche produit des données suivant un vecteur $\overrightarrow{x}$, la tâche suivante pourra consommer ces mêmes données suivant un vecteur $\overrightarrow{y}$. La désignation du nombre de points est elle aussi différente, tout se fait grâce aux matrices supports qui ne sont pas comparables entre elles. L'établissement du système d'équation est donc beaucoup plus difficile. En fait, Lee et Murthy proposent de ramener le calcul de l'ordonnancement de GMDSDF à celui de MDSDF, ils suggèrent de calculer des boites englobantes pour ranger les points des treillis.

Toutes les explications suivantes ne sont valables que pour des applications bidimensionnelles. Les autres types d'applications ne sont pas traités et sont considérés comme trop complexes.

Calcul des boites englobantes : Les données produites ou consommées par chaque tâche sont « rangées » dans des rectangles à l'aide d'une fonction de « rectangularisation ». En effet, il suffit de changer le système de coordonnées d'un treillis en prenant comme base les vecteurs de la matrice exemple[3].

Mais pour le « décimateur » et l'« expandeur », le problème est différent. Il faut seulement que ces derniers ajoutent ou suppriment des ponts sur le treillis ; la quantité de points prise sur un treillis pour une itération importe peu. On fixe alors que l« expandeur » consomme un rectangle $(1,1)$ (d'où $W_e = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$) ce qui donne par application de l'équation 1.3 $W_f = L$. On peut en déduire alors que le « décimateur » produit un rectangle $(L1, L2)$ avec $L1$ et $L2$ deux entiers positifs tels que $L1L2 = |det(L)|$ **[**50**]**.

Par un raisonnement similaire le « décimateur » produit un rectangle $(1,1)$ et consomme un rectangle $(M1, M2)$. Mais on ne peut avoir simplement $M1M2 = |det(M)|$ car il n'y aurait pas toujours de points à produire en raison de la forme des treillis, il faut donc appliquer d'autres conditions sur le calcul de $(M1, M2)$. En fait, Lee et Murthy prouvent qu'il existe toujours une décomposition de $|det(L)|$ qui soit valide. Il suffit de tester différentes possibilités jusqu'à ce que l'égalité suivante soit respectée après le calcul de l'ordonnancement :

$$|N(W_f)| = \left|\frac{N(W_e)}{det(L)}\right| \tag{1.4}$$

Calcul de l'ordonnancement : Finalement, il est possible de calculer l'ordonnancement de l'application. Une fois ce calcul effectué, il reste pour chaque tâche à définir la matrice support. En effet, on sait grâce aux rectangles combien de données ces tâches consomment, mais on ignore où ces données sont situées sur les treillis. Lee et Murthy prouvent dans **[**50**]** qu'il est possible d'obtenir les matrices supports automatiquement à partir du nombre de répétition d'une source. Àprès ce dernier calcul et la vérification de la validité des boîtes englobantes des « décimateurs », la modélisation de l'application et l'ordonnancement sont terminés.

[3] Les points du treillis sont obtenus par combinaison linéaire de la matrice exemple qui est obligatoirement inversible en GMDSDF ; donc elle est bien une base des points du treillis.

1.2.3.3 Modèlisation d'une application

La modélisation d'une application[4] en GMDSDF s'effectue de la manière suivante :
- création du graphe ;
- choix des matrices examples, d'« expansion » et de « décimation » pour les tâches spéciales ;
- choix des rectangles pour les tâches « ordinaires » ;
- calcul de l'ordonnancement ;
- calcul des matrices supports.

L'application est donc définie après le calcul de l'ordonnancement, ce qui est un inconvénient majeur puisque tout le travail est effectué par le modeleur.

1.2.3.4 Conclusion

Edward Lee reconnait lui même dans ses articles que l'utilisation de GMDSDF dans un environnement de développement ne serait pas sans poser de problèmes. Ainsi on peut regretter que GMDSDF oblige à définir, à un même niveau de description, les tâches spéciales de manipulation du treillis et les tâches ordinaires de manipulation des données. On peut regretter également que la manipulation des données sur un treillis se fasse de manière aussi régulière.

Nous introduisons ci-dessous un autre domaine multidimensionnel appelé ARRAY-OL qui ne présente pas ce genre d'inconvénients.

1.3 ARRAY-OL

1.3.1 Présentation

1.3.1.1 Principes

ARRAY-OL (Array Oriented Langage) est un langage spécialisé dans la description d'applications de traitement du signal systématique. Ce type d'application est caractérisé par une manipulation de grandes quantités de données qui sont traitées par un ensemble de tâches de façon régulière.

Les données sont contenues dans des tableaux, les flux de données sont eux représentés par des tableaux ayant une dimension infinie. Chaque tâche de l'application consomme un ou plusieurs tableaux en le(s) découpant en « morceaux » de même taille appelés motifs ; puis calcule à partir de ces motifs d'autres motifs qui seront rangés dans les tableaux résultats. La chaîne se poursuit, les tableaux produits étant à leur tour consommés.

ARRAY-OL tire donc son nom du type de structure de données manipulés. Mais il est important de souligner qu'ARRAY-OL permet seulement de spécifier les dépendances de données des applications à un certain niveau de granularité. Il est ainsi possible de déduire à la fois le parallélisme de tâches et le parallélisme de données. En revanche ARRAY-OL ne constitue pas un langage de programmation complet et ne fournit même pas un modèle d'exécution. Il est cependant possible d'en extrapoler un à partir d'une description, mais ce modèle n'est pas explicite et est induit par la description.

[4] Un exemple est présenté dans la section 1.4.1.3 page 33.

ARRAY-OL fournit donc un langage standardisé de spécification d'applications de traitement du signal intensif. À partir d'une telle spécification, il est naturel de chercher à exécuter l'application correspondante. On peut pour cela produire, à partir d'une description ARRAY-OL, un code source dans un langage cible du type (C,C++, ...) qu'on exécutera sur la plate-forme de son choix. Cette phase de transcription de l'application est appelée « compilation », ce terme est employé par abus de langage et ne doit pas induire le lecteur en erreur. Il s'agit en fait d'une « traduction » de la description dans un langage de programmation et ce n'est que la compilation de ce dernier qui permettra l'exécution.

Cette « traduction » ne peut se faire que si certaines contraintes sont respectées, ces dernières peuvent être liées :

1. au modèle d'exécution qu'il soit de type SDF du type *process network* ou du type induit par la description ;
2. à l'architecture, par exemple la quantité de mémoire disponible ;
3. à l'application elle-même et à la nécessité de gérer de façon particulière les flots de données.

En effet, l'application est définie de manière statique et il est donc nécessaire de la modifier si la traduction n'est pas possible. Par exemple, si la taille des tableaux produits à un moment t de l'exécution dépasse la taille de la mémoire disponible sur l'architecture cible, il est alors nécessaire d'opérer de telles modifications.

Ainsi, il existe un formalisme appelé ODT (Opérateurs de Distribution de Tableaux) qui sert à manipuler les dépendances de données au sein d'une tâche ARRAY-OL. À l'aide de ce formalisme, plusieurs opérations élémentaires de transformation ont été mises au point dans le but de répondre aux attentes de la compilation. Ces opérations permettent non seulement la compilation mais également l'optimisation de l'application. Toutefois nous verrons qu'elles possèdent aussi plusieurs inconvénients.

Pour conclure, nous pouvons présenter ARRAY-OL comme étant un simple langage de description de dépendances pour lequel ont été créés des outils permettant d'adapter l'application décrite à un certain nombre de critères et de contraintes en vue de son exécution.

1.3.1.2 Historique

Inventé par Alain Demeure, ARRAY-OL **[**21**]** a vu le jour chez TUS (THALES Underwater System) en 1995.

À l'origine, le premier compilateur dédié à ARRAY-OL, développé chez TUS ne supportait qu'un sous ensemble du langage. TUS a alors collaboré avec Corine Ancourt et François Irigoin[5] pour permettre une exécution SPMD d'ARRAY-OL telle quelle soit capable de satisfaire un ensemble de contraintes dynamiques et physiques. Les résultats de ces travaux sont étudiés plus en détails dans la section 2.2 page 41.

Cependant, c'est lors de la collaboration de TUS avec l'equipe WEST et notamment durant la thèse de Julien Soula **[**58**]** qu'a été développée une première version complète du compilateur. Julien Soula ne s'est pas contenté d'écrire le compilateur, il a finalisé le formalisme ODT et l'a utilisé afin de créer quatre transformations permettant la compilation d'ARRAY-OL en fonctions de différentes contraintes. Cette thèse a marqué le début de la collaboration entre TUS et l'équipe WEST.

[5] du centre de recherche en informatique de l'école des mines de Paris

À cette période, l'équipe WEST a conçu un environnement complet de spécification et de compilation dédié à ARRAY-OL baptisé GASPARD [9]. Puis de 2001 à 2003, TUS et le LIFL se sont retrouvés au sein du projet européen SOPHOCLES qui avait pour but *une validation conceptuelle d'une méthodologie, de plates-formes et de technologies supportant l'intégration, la validation et la programmation de systèmes complexes composés de composants virtuels hétérogènes dans un environnement distribué. Cette méthodologie devant permettre la création de « cyber-entreprises » fournissant des services d'intégration via le web.*

ARRAY-OL a ainsi été proposé comme une des nouvelles technologies servant de support à SOPHOCLES. THALES Communications, partenaire de SOPHOCLES et utilisateur d'ARRAY-OL a entamé à l'occasion de ce projet une collaboration de deux ans avec l'équipe WEST autour de la compilation d'ARRAY-OL.

Ces collaborations ont permis des échanges fructueux entre les besoins concrets des industriels et les compétences plus théoriques de l'équipe west. Ainsi, pour l'équipe WEST, il a été possible de travailler sur des applications réelles telles que la VBL ou la FRN, d'accroître l'interopérabilité de ses outils et de fournir en retour une expertise théorique sur la compilation d'ARRAY-OL à partir principalement des travaux de Julien Soula.

Parallèlement à cette collaboration, j'ai mené les travaux qui ont conduit à l'écriture de cette thèse. Mais avant de présenter les résultats obtenus, il est primordial de bien étudier le fonctionnement d'ARRAY-OL car sous des dehors simples voire simplistes, ARRAY-OL recèle quelques subtilités qu'il est préférable de savoir manier.

1.3.2 Fonctionnement d'ARRAY-OL

La modélisation d'une application en ARRAY-OL est découpée en deux parties qui correspondent chacune à un niveau de description : le niveau global et le niveau local. Le premier établit l'enchaînement des tâches nécessaires au fonctionnement de l'application et le second définit comment doit s'exécuter chaque tâche. Nous allons donc étudier successivement ces deux niveaux.

1.3.2.1 Modèle global

Le modèle global est un graphe orienté où les sommets sont des tableaux ou des tâches et où les arêtes indiquent le sens de l'exécution. Chaque tâche consomme un ou plusieurs tableaux, effectue un traitement quelconque sur leurs éléments et produit un ou plusieurs tableaux résultats. Il n'y a donc pas de corrélation entre le nombre de tableaux d'entrées et le nombre de tableaux de sorties.

Mais il n'y a pas non plus de corrélation entre les nombres de dimensions de ces tableaux. Ainsi il est possible pour une tâche de consommer deux tableaux bidimensionnels et de produire un tableau tridimensionnel. La création de dimensions est très utile, par exemple dans le cas d'une FFT qui crée une dimension fréquentielle. Cependant il ne peut y avoir qu'une seule dimension de taille infinie par tableau, ce qui est suffisant car cette dimension ne sert généralement qu'à représenter le temps. Enfin, une dernière spécificité des tableaux utilisés dans ARRAY-OL, c'est qu'ils sont considérés comme toriques et que donc la consommation ou la production de leurs éléments peut se faire modulo à leur taille.

Une limitation du modèle global est que le graphe ne peut pas inclure de cycles. À chaque étape de notre cheminement, nous verrons qu'ARRAY-OL comporte des limitations ; afin

de faciliter la relecture, j'ai résumé l'ensemble de ces contraintes à la fin de document en annexe A page 123). La figure 1.13 montre un exemple de niveau global.

Si on se place du point de vue de l'exécution, il est très important de comprendre que le modèle global est un graphe de dépendances et non un graphe à flots de données. Ainsi, à chacune de ses exécutions une tâche consomme un seul tableau d'entrée sur chacune de ses arêtes d'entrées et produit un seul tableau de sorties pour chacune de ses arêtes de sorties. Il est également possible d'obtenir un ordonnancement basique à partir du modèle global. Mais on ne bénificie pas de l'expression du parallélisme de données qui se fait au niveau du modèle local.

1.3.2.2 Modèle local

Le modèle local définit l'intéraction entre une tâche et ses tableaux opérandes et résultats. Il est défini sous forme d'un graphe où chaque tableau opérande et résultat est relié à la tâche de ce modèle. La façon dont cette dernière consomme et produit les tableaux qui lui sont associés peut être analysée à travers chaque couple tâche - tableau. De tels couples sont appelés demi-tâche. Un modèle local comporte également un *tiler* par demi-tâche (*cf* figure 1.14), nous découvrirons un peu plus loin le rôle de ces *tilers*.

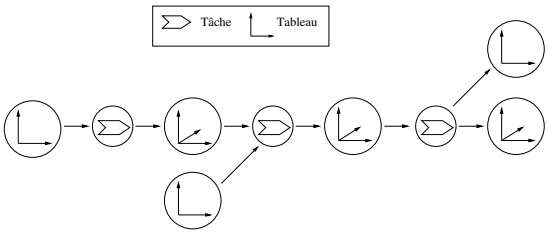


FIG. 1.13: Le niveau global

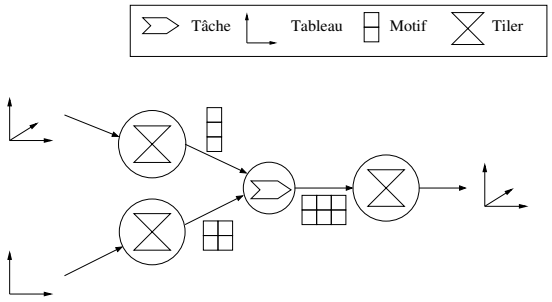


FIG. 1.14: Le niveau local

Notion de motif : soit un couple tâche ($T1$) - tableau ($A1$), si $A1$ est un tableau d'entrée, $T1$ va alors prendre un sous ensemble fini des éléments de $A1$, puis va effectuer un traitement dessus . De façon similaire, si $A1$ est un tableau de sortie alors $T1$ va lui fournir un ensemble fini d'éléments qu'elle vient de calculer. Ce sous-ensemble est appelé **motif** *(pattern)* ($M1$).

Puis l'opération recommence, $T1$ va choisir ou fournir un nouveau motif, mais ce dernier devra être obligatoirement de forme identique au premier. La forme d'un motif est donc fixée pour chaque demi-tâche (*cf* figure 1.15).

À chaque couple tâche - tableau est également associé un *tiler*. Ce dernier contient les informations nécessaires pour répondre aux questions suivantes : comment le motif est-il construit ?, comment la tâche passe-t-elle d'un motif au suivant ?

Ces informations sont :

- $\overrightarrow{o}$: la position du premier élément du premier motif ;
- $\overrightarrow{d}$: la taille des dimensions du motif ($M1$) ;
- P : la matrice de pavage ;
- F : la matrice d'ajustage ;

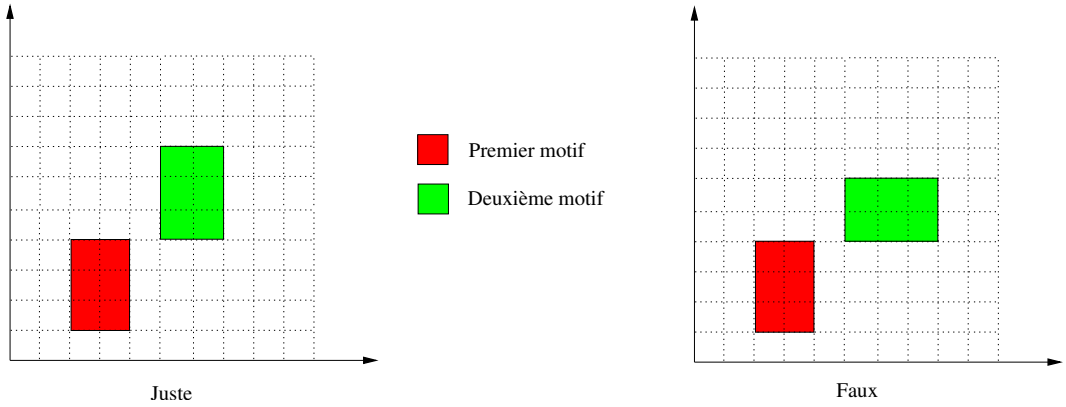


FIG. 1.15: Deux exemples de motifs

– $\overrightarrow{m}$: la taille des dimensions du tableau ($A1$).

L'utilisation des ces données repose sur les notions d'ajustage et de pavage.

Notion de pavage : pour énumérer les motifs, chaque demi-tâche dispose via son *tiler* d'une matrice de vecteurs de **pavage** *(paving)*) et d'un point de départ appelé **origine** *(origin)*. À partir de ces deux éléments, il est possible de calculer les premiers points de chaque motif. Les coordonnées de ces points sont calculées comme la somme des coordonnées de l'origine et d'une conbinaison linéaire des vecteurs de pavage, le tout modulo la taille du tableau puisque les tableaux ARRAY-OL sont toriques[6] (*cf* équation 1.5).

$$\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{Q}, \overrightarrow{r_q} = (\overrightarrow{o} + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \tag{1.5}$$

Exemple : Si on dispose d'un tableau à deux dimensions avec comme point d'origine $\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ et comme matrice de pavage $\left(\begin{smallmatrix}2&0\\0&3\end{smallmatrix}\right)$ alors les motifs commenceront aux points montrés sur la figure 1.16 [7].

Notion d'ajustage : Grâce au pavage, nous connaissons le premier point de chaque motif ($\overrightarrow{r}$) et grâce au *tiler*, nous connaissons la taille de chacune de ses dimensions ($\overrightarrow{d}$). Il est facile de penser qu'il suffit de positionner le motif sur chacun des premiers points et de prendre comme autres points les éléments du tableau ainsi délimité.

Mais il n'en est rien et ceci pour deux raisons :

– premièrement parce que deux éléments consécutifs du motif ne sont pas forcément contigus sur le tableau (*cf* figure 1.18 page 24). Pour passer d'un élément à un autre, nous utiliserons une matrice d'**ajustage** *(fitting)* ;

[6] $\overrightarrow{Q}$ est le vecteur des bornes de pavage, son calcul est expliqué section 1.3.2.3 page 26 , car il ne dépend pas d'une seule relation tâche - tableau, mais de toutes ces relations.

[7] La figure montre les points de départ des différents motifs obtenus par itération sur la matrice de pavage. Mais l'ordre d'obtention de ces points ne constituent en rien une obligation pour un quelconque modèle d'exécution.

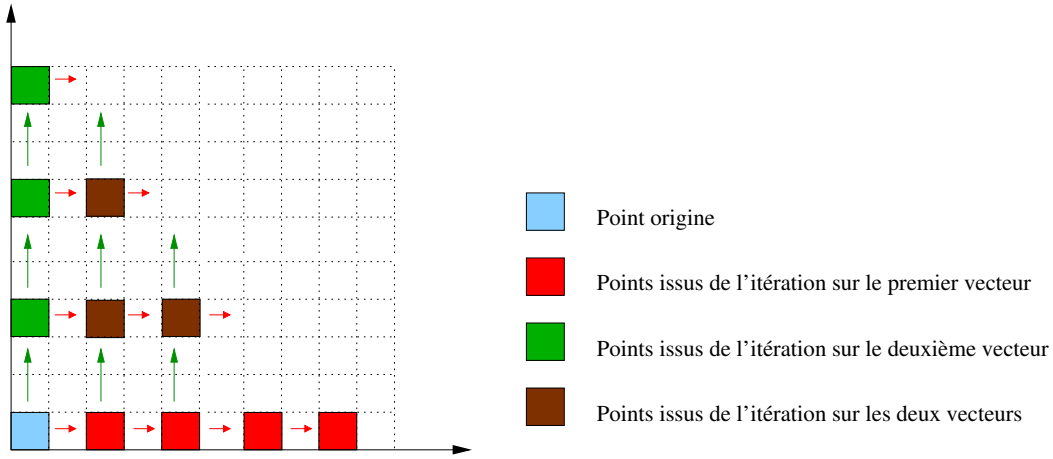


FIG. 1.16: Exemple d'itérations

– deuxièmement parce que la forme du motif, c'est-à-dire son nombre de dimensions, ne correspond en rien à celui du tableau. En effet, un motif peut avoir plus, autant ou moins de dimensions que le tableau (*cf* figure 1.18).

Ainsi, l'utilisation de la matrice d'ajustage s'effectue de manière similaire à celle de la matrice de pavage. Les éléments d'un tableau constituant un motif sont calculés comme la somme des coordonnées du premier élément de ce motif et d'une combinaison linéaire de la matrice d'ajustage, le tout modulo la taille du tableau puisque les tableaux ARRAY-OL sont toriques (*cf* équation 1.6).

$$\forall \overrightarrow{x_d}, \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{D}, (\overrightarrow{r_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m} \tag{1.6}$$

Exemples : Quelques exemples simples sont donnés sur la figure 1.17 : le dernier de ces exemples montre qu'il ne faut pas confondre le motif et la boîte englobante des éléments constituant ce motif, un motif est toujours compact.

La figure 1.18 aborde un cas plus complexe : supposons que nous ayons comme tableau un vecteur avec comme origine des itérations $\begin{pmatrix} 0 \end{pmatrix}$ et comme motif associé un tableau bidimensionnel de taille $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$. Dans le premier cas, la matrice d'ajustage est $\begin{pmatrix} 1 & 3 \end{pmatrix}$, chaque vecteur d'ajustage va servir à remplir une dimension du motif. Le nombre d'itérations sur un vecteur est égal à la taille de la dimension qu'il permet de remplir. On s'aperçoit donc qu'il est possible d'avoir plus de dimensions dans le motif que dans le tableau. Dans le deuxième cas, la matrice d'ajustage est $\begin{pmatrix} 2 & 6 \end{pmatrix}$, en déroulant les itérations, on voit que les éléments du motif ne sont pas issus d'éléments consécutifs du tableau.

Enfin, dernière idée préconçue dont il faut se méfier, les éléments du tableau constituant le motif ne forment pas nécessairement une forme géométrique régulière (un parallélépidède n-dimensionel) comme le montre la figure 1.19 . On peut en complexifiant la matrice d'ajustage atteindre des formes quasi circulaires. Cependant, il ne s'agit que de cas d'école, les applications visées par ARRAY-OL ne nécessitant pas de formes aussi complexes.

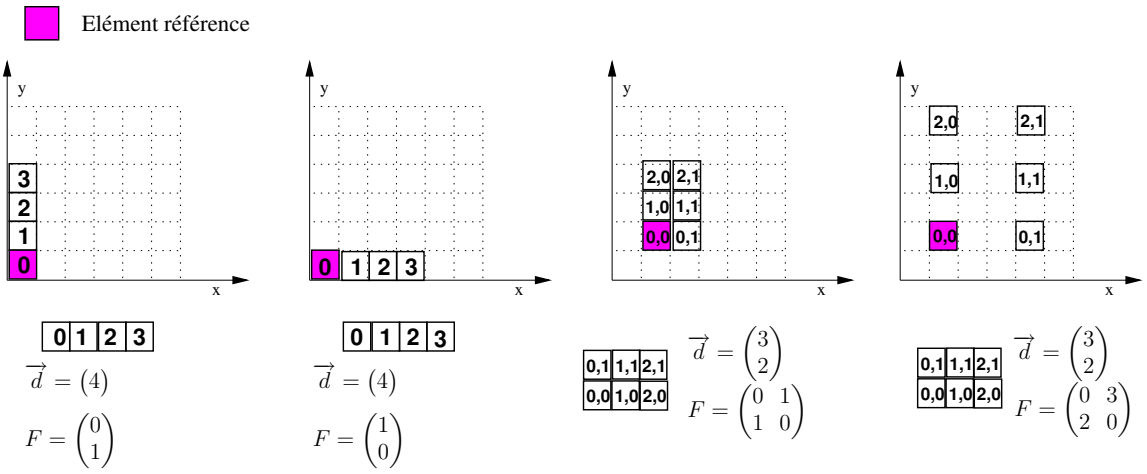


FIG. 1.17: Exemples simples d'ajustage

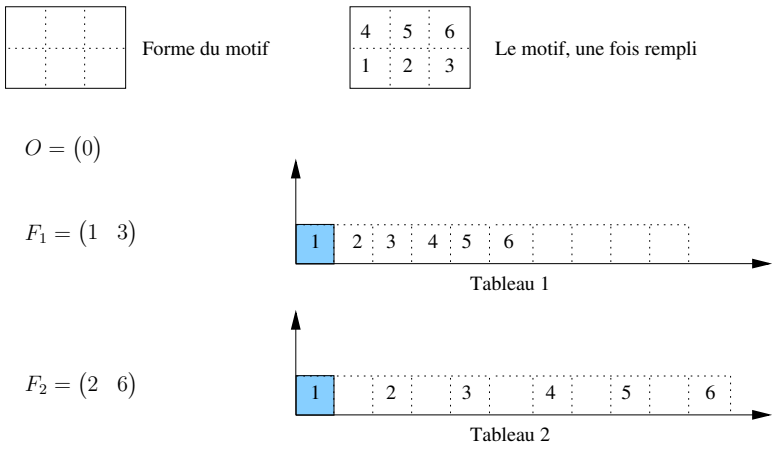


FIG. 1.18: Exemples d'ajustage

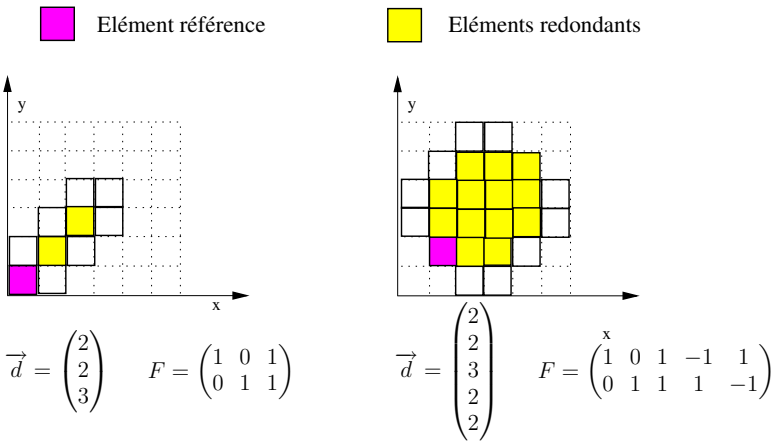


FIG. 1.19: Exemples complexes d'ajustage

Conclusion : résumons nous, pour définir une relation tâche - tableau, il faut l'origine, la matrice de pavage, le motif et la matrice d'ajustage. L'interaction entre ces données est très simple : on prend l'origine comme point de départ, on itère les vecteurs d'ajustage pour construire le motif que l'on passe à la tâche, puis on itère les vecteurs de pavage pour connaître le nouveau point de départ. La seule limitation induite par ce système est que les tâches doivent consommer et produire les tableaux de manière régulière avec des motifs de taille constante. Toutes ces explications tiennent en deux équations. L'équation 1.7 donne les coordonnées des origines du motif et l'équation 1.8 donne l'ensemble des coordonnées des points du motif pour l'itération X_q.

$$\forall \overrightarrow{X_q},\ \overrightarrow{0} \leq \overrightarrow{X_q} < \overrightarrow{Q}, (\overrightarrow{O} + P \times \overrightarrow{X_q}) \mod \overrightarrow{m} \tag{1.7}$$

$$\forall \overrightarrow{X_d},\ \overrightarrow{0} \leq \overrightarrow{X_d} < \overrightarrow{D}, (\overrightarrow{O} + P \times \overrightarrow{X_q} + F \times \overrightarrow{X_d}) \mod \overrightarrow{m} \tag{1.8}$$

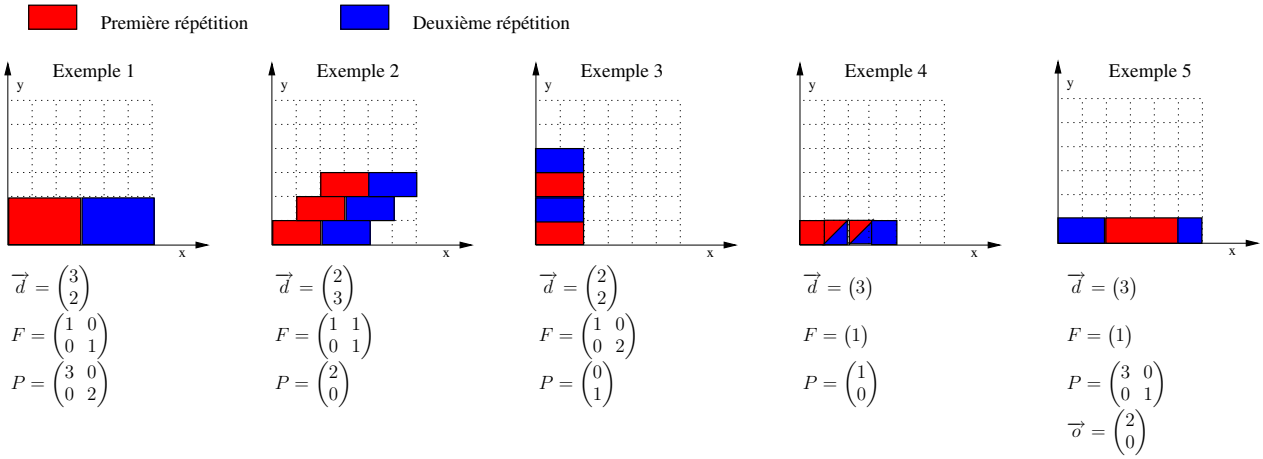


FIG. 1.20: Différentes répétitions

La figure 1.20 illustre la combinaison du pavage et de l'ajustage en montrant deux répétitions successives.

1. Un cas basique.
2. Les consommations des motifs ne sont pas nécessairement parallèles aux axes.
3. Deux motifs successifs qui s'entrecroisent.
4. Deux motifs successifs qui se chevauchent.
5. Un tableau torique avec une origine qui n'est pas à $\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ mais à $\left(\begin{smallmatrix}2\\0\end{smallmatrix}\right)$.

Connaissant les relations entre la tâche et chacun de ses tableaux, il nous faut découvrir comment ces relations interagissent, mais également comment elles permettent de déterminer les bornes du pavage.

1.3.2.3 Vision globale de la tâche.

Nous nous intéressons dorénavant à la tâche dans sa globalité. À ce niveau, il existe une corrélation entre tous les pavages . En effet la tâche pour réaliser le traitement qui est le sien

à besoin d'un motif de chaque tableau opérande pour produire un motif de chaque tableau résultat. Il y a donc partout le même nombre d'itérations de pavage et par conséquent le même nombre de vecteurs. Les itérations de pavage jouent le rôle de « métronome » dans une tâche, chacune marquant une exécution de la tâche. Il est donc possible d'associer à chaque itération les motifs produits et consommés par la tâche.

Critères d'arrêt. Le calcul du nombre d'itérations est problématique, les tableaux étant toriques il est possible de les parcourir indéfiniment. Cependant il suffit de produire l'intégralité des résultats pour considérer que le rôle de la tâche est terminé. En effet, ARRAY-OL est un langage à assignation unique : les éléments des tableaux résultats sont calculés une seule fois.

Donc, essayons de calculer les bornes de pavage à l'aide d'un tableau résultat. Il suffit de trouver le nombre d'itérations minimum pour le remplir. Mais nouveau problème, le calcul n'est pas linéaire à cause du modulo. Ceci va entraîner une nouvelle simplification du modèle qui consiste à interdire le modulo pour les tableaux résultats. Il en découle que les vecteurs de pavages sont nécessairement parallèles aux axes et par construction que les vecteurs d'ajustages le sont aussi.

Le calcul des bornes est ainsi rendu possible et il est décrit en détail dans la section 1.3.2.3. Toutefois ce n'est pas tout : les vecteurs d'ajustage et de pavage doivent être positifs. Ceci peut s'expliquer par l'obligation d'avoir cette propriété pour certaines démonstrations. Il en résulte que l'origine est forcement le premier élément du tableau. Il est à noter qu'une borne peut être infinie si elle s'applique à un vecteur pavant une dimension infinie.

Une demi-tâche répondant à tous ces critères est dite « **exacte** ». Celle choisie pour effectuer le calcul doit être marquée comme « **maître** » *(master)* dans la description ARRAY-OL et il s'agit obligatoirement d'une demi-tâche résultat.

Exemple. Nous allons illustrer par l'exemple du produit de matrice la nécessité d'avoir le même nombre d'itérations de pavages pour toutes les demi-tâches. Soit $A1$ une matrice 3 x 5 et $A2$ une matrice 5 x 2, on calcule le produit $A1 \times A2 = A3$ avec $A3$ de taille 3 x 2. La figure 1.21 montre la description de cette tâche.

Cet exemple montre la nécessité pour une tâche d'avoir le même nombre de vecteurs et les mêmes bornes pour le pavage alors que le produit de matrices ne se conçoit pas a priori avec deux vecteurs de pavage par tableau opérande. D'ailleurs on peut voir l'itération sur un vecteur comme une « astuce ».

Cet exemple montre, en outre, une limitation majeure d'ARRAY-OL : il n'est possible de traiter uniquement que des matrices de taille connue statiquement. Ce produit n'est donc pas générique et il est nécessaire d'en spécifier un nouveau pour multiplier deux matrices de tailles différentes. (Les bornes d'ajustage ne seraient plus les mêmes).

Calcul du Critère d'arrêt. Le calcul des bornes des itérations de pavage ($\overrightarrow{Q}$) s'effectue de la manière suivante :

- on classe les vecteurs de pavage par groupe de vecteurs colinéaires (ce qui correspond à un groupe par dimension) ;
- on trie par ordre croissant chacun de ces groupes en fonction de la norme des vecteurs ;
- finalement, on obtient la borne de pavage d'un vecteur en divisant la norme de son successeur par la propre norme de ce vecteur. Le plus grand vecteur d'un groupe uti-

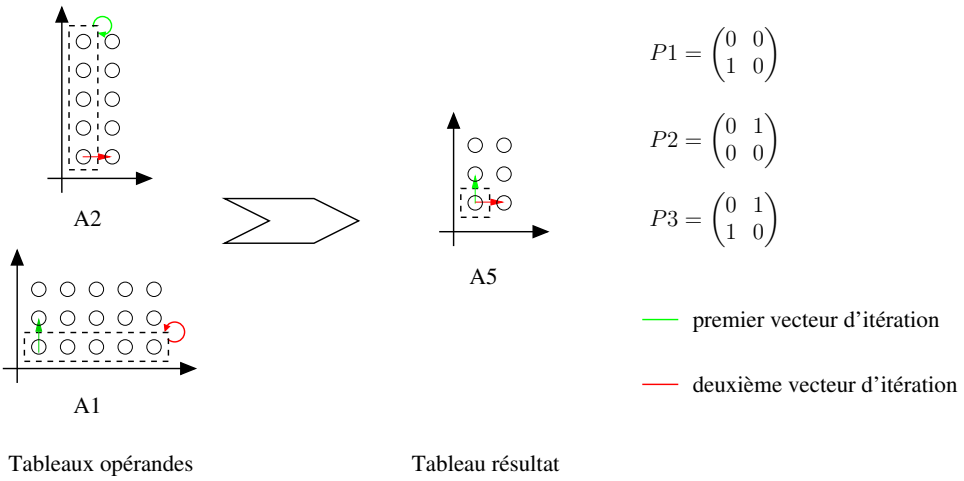


FIG. 1.21: Le produit de matrice

Calcul des bornes d'itération de pavage

Soit un tableau tri-dimensionnel de taille : $\begin{pmatrix} 64 \\ 200 \\ 125 \end{pmatrix}$ et sa matrice de pavage associée : $\begin{pmatrix} 32 & 0 & 1 & 0 & 0 & 0 & 8 & 0 \\ 0 & 1 & 0 & 0 & 50 & 5 & 0 & 0 \\ 0 & 0 & 0 & 25 & 0 & 0 & 0 & 1 \end{pmatrix}$. Pour calculer les bornes associées aux vecteurs, on procède en 3 étapes :

– On groupe les vecteurs colinéaires : $\begin{pmatrix} 32 & 1 & 8 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 50 & 5 \\ 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 25 & 1 \end{pmatrix}$

– On les classe ensuite par ordre croissant en fonction de leur norme : $\begin{pmatrix} 1 & 8 & 32 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 5 & 50 \\ 0 & 0 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 25 \end{pmatrix}$

– On effectue les divisions des normes pour obtenir les bornes : $\begin{pmatrix} 8 & 4 & 2 \end{pmatrix}$ $\begin{pmatrix} 5 & 10 & 4 \end{pmatrix}$ $\begin{pmatrix} 25 & 5 \end{pmatrix}$

On obtient ainsi les bornes de pavages qu'il faut réassocier à chaque vecteur.

Finalement $Q = \begin{pmatrix} 2 \\ 5 \\ 8 \\ 5 \\ 4 \\ 10 \\ 4 \\ 25 \end{pmatrix}$

lise la taille de la dimension qui correspond à ce groupe comme norme du vecteur suivant.

Un exemple de mise en œuvre de cet algorithme est donné dans l'encadré page 27.

1.3.2.4 Tâches hiérarchiques

Pour l'instant nous avons simplement considéré les tâches comme de simples « boîtes noires » qui effectuent un traitement prédéterminé. Mais ce n'est pas toujours le cas, les tâches peuvent être hiérarchiques, c'est-à-dire qu'elles peuvent être elles mêmes des applications ARRAY-OL avec leur modèle local et leur modèle global. Elles prennent respectivement dans ce cas comme tableaux d'entrée et de sortie les motifs opérandes et résultats de la tâche supérieure. On distingue ainsi les tâches élémentaires (TE) et hiérarchiques. On remarquera que la forme des motifs peut amener à avoir des tâches hiérarchiques qui consomment des tableaux à n dimensions dans la partie supérieure et à m dimensions dans la sous-tâche.

1.3.3 Modèlisation d'une application

Nous venons de voir les principes théoriques d'ARRAY-OL. Ces principes ne sont pas compliqués, cependant la modélisation d'une application en ARRAY-OL n'est pas nécessairement simple.

Le problème vient principalement du remplissage des *tilers*, il faut indiquer pour chacun : l'origine, les dimensions du motif, la matrice de pavage et la matrice d'ajustage. En dehors du fait que cette étape peut s'avérer longue et fastidieuse, l'élaboration ou même l'interprétation des matrices de pavage et d'ajustage reste un passage difficile surtout pour le néophyte.

Mais il est heureusement possible de vérifier la validité des *tilers* par l'utilisation d'égalités mathématiques simples et par le truchement de la représentation graphique des données qu'ils contiennent.

1.3.3.1 Égalités mathématiques

À partir de la définition des constituants du *tiler*, il est possible d'établir un certain nombre d'égalité permettant de vérifier rapidement la validité « syntaxique » de ces constituants.

- le nombre de ligne des matrices de pavage et d'ajustage (P et F) et le nombre de dimensions du vecteur origine ($\vec{O}$) doivent être égaux au nombre de dimensions du tableau ($\vec{m}$) ;
- le nombre de colonnes d'une matrice d'ajustage (F), c'est-à-dire le nombre de vecteurs dont elle est constituée, doit être égale au nombre de dimensions du motif ($\vec{d}$) [8] ;
- le nombre de colonnes des matrices de pavage (P), c'est à dire le nombre de vecteurs dont elles sont constituées, doit être le même pour toutes les matrices d'un même modèle local ;
- il ne doit y avoir qu'une seule demi-tâche maîtresse par modèle local ;
- dans le cas d'une demi-tâche maîtresse, l'origine doit être le point de coordonnées ($\vec{0}$) et chaque vecteur de pavage et d'ajustage ne doit contenir qu'un seul élément non nul.

La vérification effectuée n'est donc que « syntaxique », rien ne prouve en effet que ces matrices désignent les points que le concepteur de l'application voulait référencer.

1.3.3.2 ARRAY-OL exemple

Afin d'effectuer une vérification « sémantique », j'ai proposé l'élaboration d'un logiciel facilitant le remplissage des *tilers*. Ce logiciel, nommé « ARRAY-OL exemple », guide pas à pas le modeleur dans la saisie des différentes données et permet à chaque étape de visualiser dans un environnement 3D le résultat de la saisie. Bien que plusieurs études **[**8, 22**]** aient été lancées dans l'équipe WEST sur ce sujet, l'implémentation proposée ici est la première à être réellement fonctionnelle.

Afin de permettre la réalisation de ce logiciel, j'ai encadré plusieurs stages en master 1 et 2. La version finale qui est actuellement redistribuée sur le site[9] de l'équipe WEST a été réalisée par Jérome Coppens et Christofle Mouflin.

[8] Seul cas particulier, si le motif est constitué d'un seul élément, il n'y a pas de matrice d'ajustage.

[9] http ://www.lifl.fr/west/aoltools/

Son fonctionnement repose sur la saisie des données directement grâce à un environnement de forme vectorielle ou matricielle. Cet environnement s'adapte automatiquement en fonction des informations préalablement entrées. Ainsi, il faut d'abord saisir les dimensions du tableau, ce qui indique donc le nombre de lignes pour les matrices de pavage et d'ajustage. Le nombre de dimensions du motif indique lui le nombre de colonne pour la matrice d'ajustage. On peut de cette façon appliquer toutes les règles de la vérification « syntaxique ».

De plus, dès que l'utilisateur valide une donnée, il visualise immédiatement dans l'environnement 3D l'effet produit. La figure 1.22 montre la saisie d'une matrice d'ajustage, la partie gauche représente les élements du tableau désignés par la matrice et la partie droite représente le motif. La figure 1.23 montre l'ensemble des éléments constituant les différents motifs (les origines sont en rouge vif).

Toutefois, il n'est pas possible de visualiser des tableaux à plus de trois dimensions. Il serait cependant envisageable de contourner le problème en proposant une projection paramétrable des dimensions.

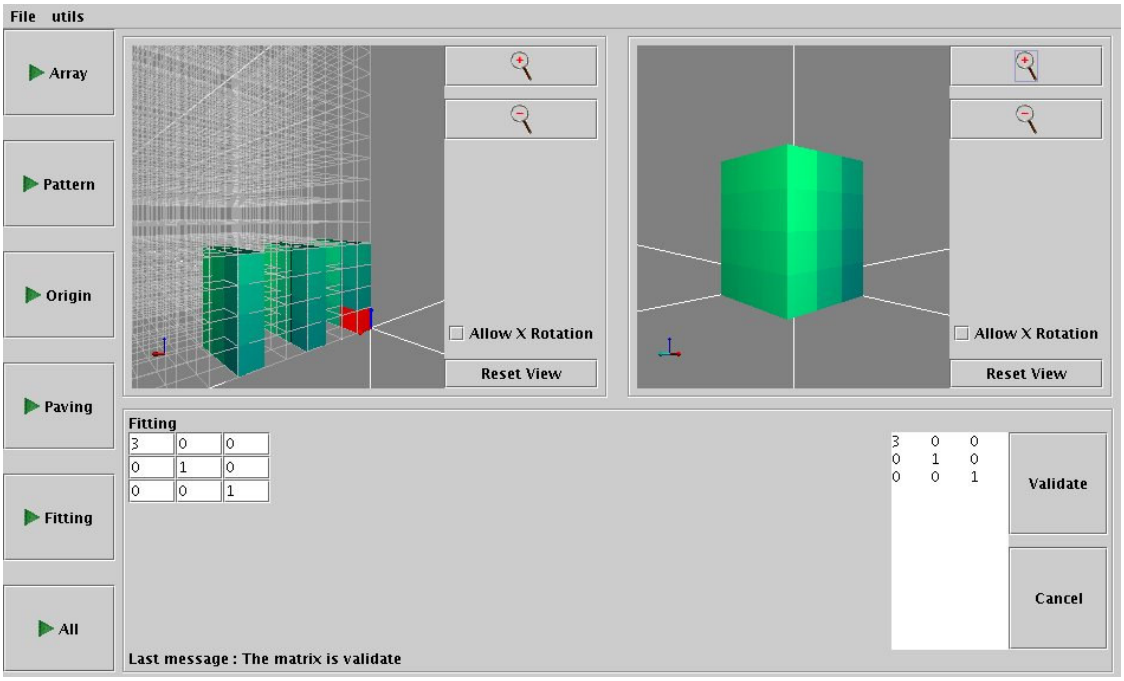


FIG. 1.22: Saisie de l'ajustage

1.3.4 Modèles d'exécution

Nous allons maintenant analyser les dépendances qui régissent l'exécution d'une application et aussi suivant quels modèles cette exécution peut se dérouler.

Une première méthode est d'employer un modèle totalement séquentiel dans lequel les tâches sont exécutées les unes après les autres et où pour chaque tâche les motifs sont également calculés successivement. Ce modèle a deux problèmes majeurs :

– les flux infinis bloquent le déroulement de l'exécution, en effet la première tâche qui a une borne de pavage infinie sera exécutée sans fin ;

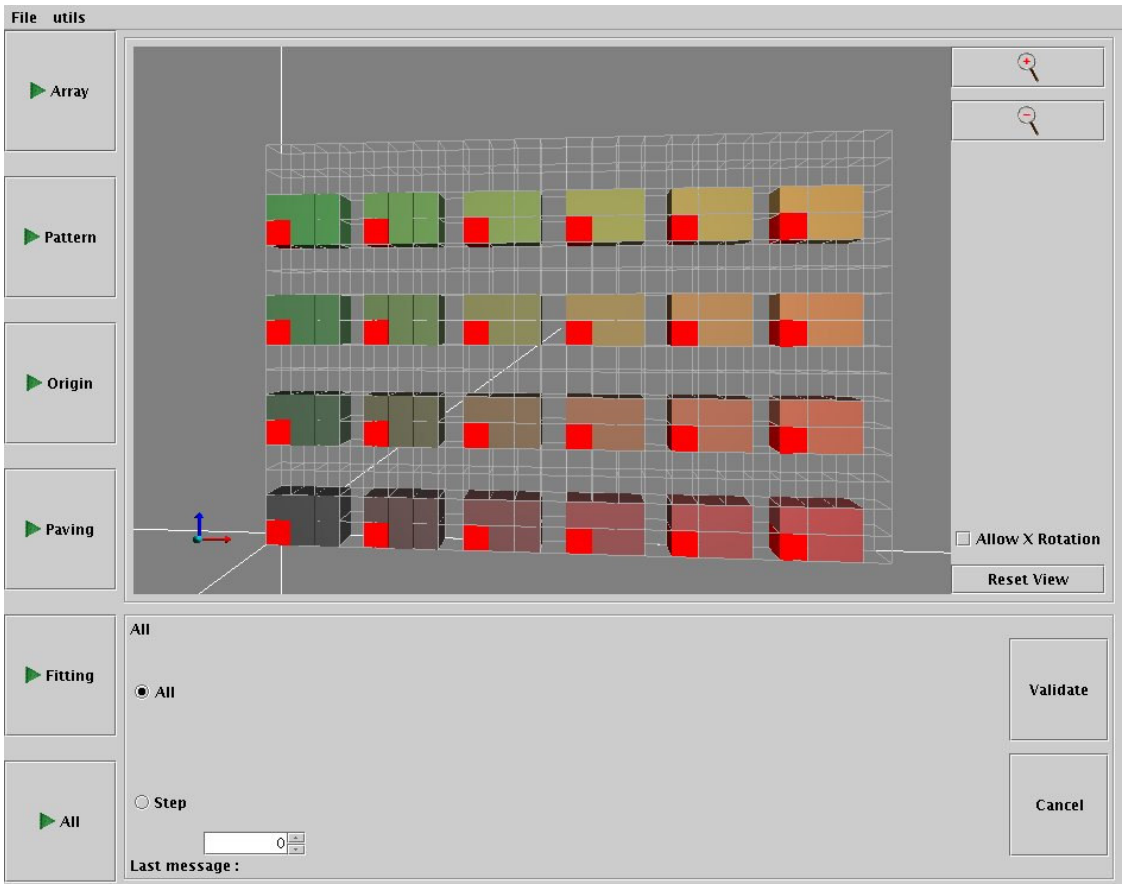


FIG. 1.23: Ensemble des motifs

– dans ce modèle, tous les tableaux intermédiaires sont produits, ils occupent donc de la place en mémoire, ce qui est particulièrement vrai dans le cadre du traitement du signal.

Un modèle d'exécution parallèle ne change pas grand chose bien qu'il bénéfice du fort degré de parallélisme exprimable par ARRAY-OL. En effet seule l'exécution au sein d'une tâche est accélérée. Mais il n'est pas possible d'exécuter simultanément deux tâches qui ont un ou des tableaux communs. En effet les motifs résultats de la première ne sont pas forcement identiques aux motifs opérandes de la deuxième (*cf* figure 1.24).

La solution serait de ne produire que le nombre de motifs résultats nécessaires pour exécuter au moins une itération de la tâche suivante. Ainsi les tableaux intermédiaires entre ces deux tâches ne seraient plus stockés dans leur intégralité, on gagnerait de la place mémoire et on résoudrait le problème du flux. De plus on gagnerait un niveau de parallélisme car il serait possible de calculer simultanément plusieurs motifs. On aurait ainsi un modèle mi-pipeline mi-data-parallèle. L'obtention d'un tel modèle est décrit dans le chapitre 4.2 page 99.

Cependant on peut d'ores et déjà objecter que dans une configuration telle que la succession de deux produits de matrice, la production du premier résultat ligne par ligne et sa consommation colonne par colonne bloque toute velléité de mise en pipeline. Cette situation problématique est nommée *corner turn* (*cf* figure 1.24).

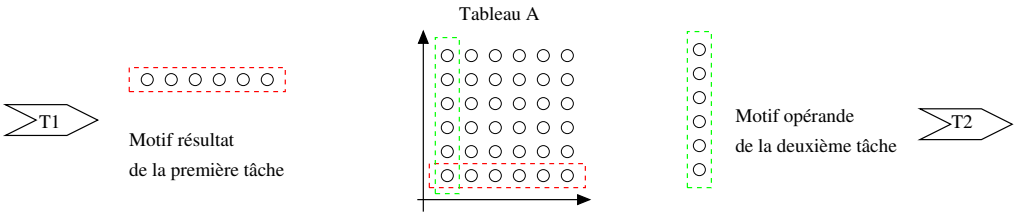


FIG. 1.24: Exemple de « *corner turn* »

1.3.5 Conclusion

La présentation que nous venons de faire montre qu'ARRAY-OL est parfaitement bien adapté à la modélisation des applications du traitement du signal systématique. Cependant l'absence de modèle de calcul ou de formalisme mathématique laisse entrevoir la difficulté de la compilation.

1.4 Comparaison d'ARRAY-OL et de GMDSDF

Comme nous venons de le voir GMDSDF et ARRAY-OL sont les seuls modèles existants permettant une description de haut niveau des applications multidimensionnelles pour le traitement du signal systématique. Le lecteur aura pu se faire une première idée sur ces deux modèles grâce à la lecture des sections précédentes, mais leur complexité n'aide pas à se faire une idée concrète de leur utilisation. C'est pourquoi nous proposons ici, une étude comparée de ARRAY-OL et de GMDSDF que se soit du point de vue pratique que du point de vue théorique.

1.4.1 Comparaison sur un exemple

Pour illustrer l'utilisation de GMDSDF et ARRAY-OL nous allons étudier la modélisation d'une application type. Le choix de l'application s'est porté sur un exemple utilisé par Lee et Murthy dans leurs articles **[**49, 51, 50**]** parlant de GMDSDF. Cet exemple est lui-même issu d'un autre article traitant de la conversion de signal video **[**44**]**.

1.4.1.1 Analyse d'un signal video

Un signal video peut aisement être considéré comme multidimensionnel, la hauteur et la largeur d'une image sont les deux premières dimensions et le temps est la troisième. L'échantillonnage d'un signal video se fait traditionnellement sur deux dimensions et garde la troisième intacte. Ainsi la longueur est gardée intacte, mais la hauteur et le temps sont échantillonnés. En effet les calculs s'effectuent toujours sur des groupes de lignes appartement à une image et sur leur évolution à travers le temps.

Nous allons donc manipuler des ensembles de lignes qui se suivent que se soit verticalement sur une même image ou temporellement sur des images qui se succèdent. La représentaion est donc bidimensionnelle avec comme abscisse le temps et comme ordonnée la hauteur d'une image. Un point du plan ainsi défini est une ligne d'une image. De part sa construction, ce plan est qualifié de « temporo-vertical ».

De plus le signal traité est un signal de télévision, sa fréquence est donc de 50Hz, il comporte 625 lignes par image et il est interlacé ce qui signifie qu'une image est en réalité constituée de deux images successives chacune comportant seulement la moitié des lignes. Ce signal est réprésenté suivant le plan « temporo-vertical » par la figure 1.25.

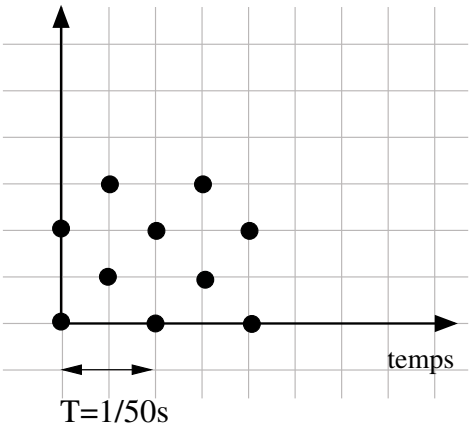


FIG. 1.25: Réprésentation d'un signal TV sur le plan « temporo-vertical »

1.4.1.2 Description de l'application

L'application qui nous concerne convertit un signal au format $4/3$ en $16/9$. La hauteur d'une image est noté P_h et sa largeur P_w ce qui donne $P_h = \frac{3}{4}P_w$ pour le format $4/3$ et ${P_h}' = \frac{9}{16}P_w$ pour le format $16/9$. On note également d_y et ${d_y}'$ la distance entre deux lignes du signal respectivement pour les formats $4/3$ et $16/9$ (*cf* figure 1.26). Comme le nombre d'image $N = 625$ reste le même dans les deux formats, on peut écrire :

$$ {d_y}' = \frac{{P_h}'}{N} = \frac{9}{16}\frac{P_w}{N} = \frac{3}{4}\frac{P_h}{N} = \frac{3}{4}d_y \tag{1.9} $$

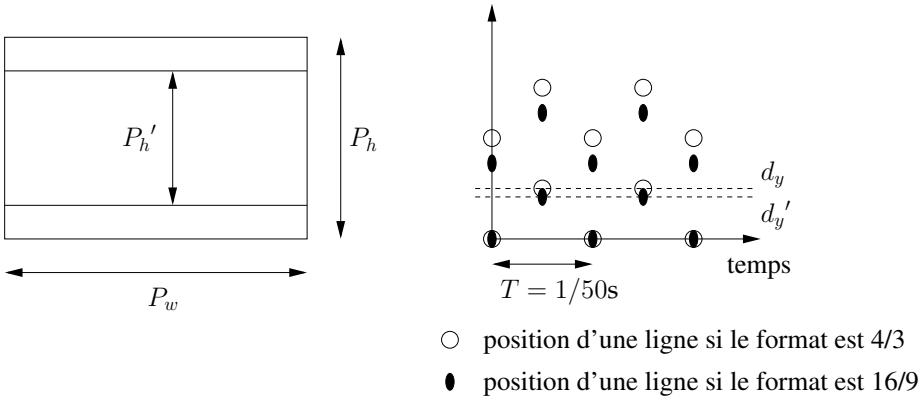


FIG. 1.26: Representation des formats $4/3$ et $16/9$ sur le plan « temporo-vertical »

Une des possibilités pour faire la conversion est de prendre $3/4$ des lignes des images au format $4/3$, de les interporler afin d'obtenir des lignes intermédiaires pour garder le nombre de lignes constant et d'obtenir ainsi les 625 lignes de l'image au format $16/9$.

Dans l'article **[**44**]**, les auteurs proposent de pratiquer cette conversion en trois étapes : deux interpolations successives, elles-mêmes suivies par un « décimateur » (*cf* figure 1.27). Les deux interpolations sont utilisées pour générer des lignes intermédiaires en fonction des lignes d'entrées. Le « décimateur » ne fait que sélectionner 625 lignes parmi toutes celles produites par les interpolations. La description de l'application présente dans l'article indique que la première interpolation consomme 11×5 lignes et en produit 4 verticalement et que la deuxième en consomme 3 verticalement et en produit 2 toujours verticalement. On apprend également que dans chaque interpolation les données consommées se recouvrent d'une itération à l'autre. « décimateur », quant à lui, ne fait que garder une ligne sur six ; mais il recrée l'interlacement en effectuant un décalage de 3 lignes une fois sur deux. L'application est représentée par la figure 1.27.

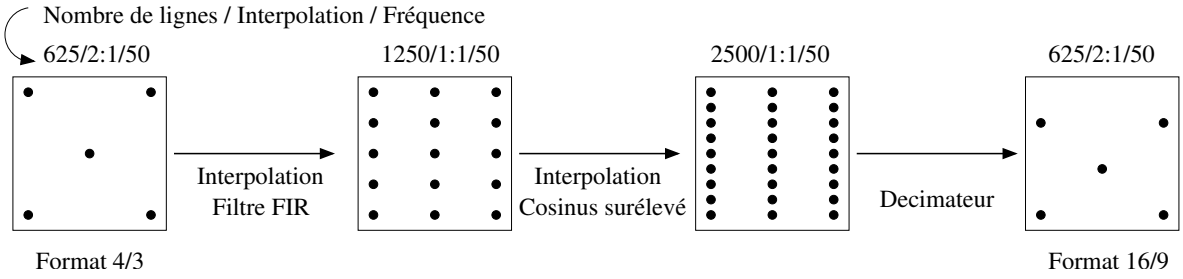


FIG. 1.27: L'application de conversion $4/3$ en $16/9$

1.4.1.3 L'application en GMDSDF

La modélisation de l'application est à priori une suite de cinq tâches : une source qui « émet » le signal, suivit de deux interpolations, puis d'un « décimateur » et enfin d'une dernière tâche qui reçoit les données. Mais comme les interpolations ajoutent des lignes et donc changent le treillis comme le montre la figure 1.27, il faut rajouter deux « expandeurs » avant chaque interpolations. La modélisation comporte désormais sept tâches. Cependant pour ne pas alourdir les calculs qui vont suivre, les deux interpolations ne sont pas représentés (*cf* figure 1.28). Si cela n'avait pas été le cas, notre modélisation comporterait simplement deux tâches supplémentaires consommant et produisant des rectangles de tailles respectives : $(11, 5)$, $(1, 4)$ et $(1, 3)$, $(2, 3)$.

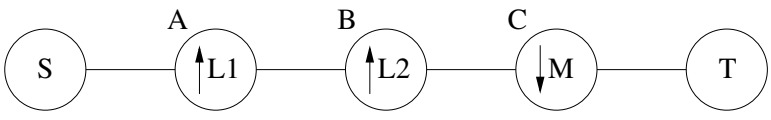


FIG. 1.28: L'application en GMDSDF

Calcul des matrices exemples : À l'aide de l'équation 1.9 page 32 et de la figure 1.27 , il est facile de déduire la valeur des matrices exemples pour les treillis du graphe. La construction de la première de ces matrices (V_{SA}) s'effectue de la manière suivante :

- on prend comme premier vecteur $\binom{2T_f}{0}$ afin de pouvoir se déplacer horizontalement d'une ligne sur deux ;
- on utilise ensuite un autre vecteur de valeur $\binom{T_f}{dy}$ pour effectuer le décalage vertical dû à l'interpolation.

En appliquant une démarche similaire, on obtient :

$$V_{SA} = \begin{bmatrix} 2T_f & T_f \\ 0 & d_y \end{bmatrix}, V_{AB} = \begin{bmatrix} T_f & 0 \\ 0 & d_y/2 \end{bmatrix}, V_{BC} = \begin{bmatrix} T_f & 0 \\ 0 & d_y/4 \end{bmatrix}, V_{CT} = \begin{bmatrix} 2T_f & T_f \\ 0 & 3d_y/4 \end{bmatrix} \quad (1.10)$$

Calcul des matrices d'« expansion » et de « décimation » : Puis en combinant les matrices exemples, que nous venons d'obtenir, à l'aide de l'équation 1.3 page 16, on calcule la valeur des matrices d'« expansion » et de « décimation ». On prend $T_f = 1$ et $d_y = 1$ pour simplifier ce qui n'affecte en rien les calculs puisque T_f et d_y servent uniquement d'unité du plan « temporo-vertical ».

$$L_1 = V_{AB}^{-1} V_{SA} = \begin{bmatrix} 1/T_f & 0 \\ 0 & 2/d_y \end{bmatrix} \begin{bmatrix} 2T_f & T_f \\ 0 & d_y \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$$
$$L_2 = V_{BC}^{-1} V_{AB} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$
$$M = V_{BC}^{-1} V_{CT} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$$

Calcul des rectangles de données : Maitenant que les matrices définissant l'application sont déterminées. Il reste à calculer les rectangles de données consommées et produites par chaque tâche afin de pouvoir obtenir ensuite l'ordonnancement de l'application. Le calcul est aisé pour les deux « expandeurs » et le « décimateur » (*cf* section 1.2.2.2 page 14).

$$|det(L_1)| = 4 = 2 \times 2, |det(L_2)| = 2 = 1 \times 2, |det(M)| = 6 = 3 \times 2$$

Seul le rectangle définissant la quantité de données émises par la source reste à calculer. Lee postule dans [50], sans aucune sorte d'explications, que la source produit toutes les points du treillis se trouvant dans les deux premières lignes et les seize premières colonnes. Par construction (*cf* figure 1.27), il est facile de voir que la source produit un rectangle de 2×8. Ce résultat est également démontrable en multipliant l'inverse de la matrice example du treillis et la matrice $\left(\begin{smallmatrix} 2 & 0 \\ 0 & 16 \end{smallmatrix}\right)$ ce qui donne la matrice support de la source ; on en déduit ensuite les points du treillis qu'elle produit. La figure 1.29 rassemble les rectangles de données.

Cependant la description de l'application n'est pas encore finie puisqu'il nous faut calculer l'ordonnacement avant d'avoir les matrices supports.

Calcul de l'ordonnancement : À l'aide de tous les rectangles, on résout le système d'équation servant à déterminer l'ordonnancement. On obtient alors le résultat suivant :

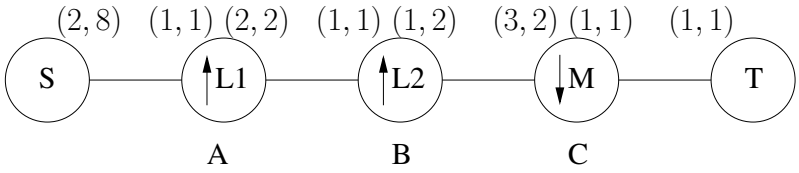


FIG. 1.29: Les rectangles de données

$$\begin{aligned} 2r_{S,1} &= 1r_{A,1}, 8r_{S,2} = 1r_{A,2} \\ 2r_{A,1} &= 1r_{B,1}, 2r_{A,2} = 1r_{B,2} \\ 1r_{B,1} &= 3r_{C,1}, 2r_{B,2} = 2r_{C,2} \\ 1r_{C,1} &= 1r_{T,1}, 1r_{C,2} = 1r_{T,2} \end{aligned}$$

Le système d'équations

$$\begin{aligned} r_{S,1} &= 3, r_{S,2} = 1 \\ r_{A,1} &= 6, r_{A,2} = 8 \\ r_{B,1} &= 12, r_{B,2} = 16 \\ r_{C,1} &= 4, r_{C,2} = 16 \end{aligned}$$

La solution de ce système

Calcul des matrices supports : Nous pouvons enfin calculer les matrices supports et vérifier, comme nous l'avons vu dans la section 1.2.3.2 page 17, que le choix du rectangle de données pour le « décimateur » était correct. Le calcul des matrices supports s'effectue en utilisant l'équation 1.3 page 16 avec la matrice support de la source. Dans notre exemple, nous avons dit que la source produisait des points dans le rectangle $\left(\begin{smallmatrix} 2 & 0 \\ 0 & 16 \end{smallmatrix}\right)$, on peut alors écrire en se référant à l'équation 1.3 page 16 :

$$W_{SA} = V_{SA}^{-1} \begin{bmatrix} 2 & 0 \\ 0 & 16 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 16r_{S,2} \end{bmatrix} = \begin{bmatrix} r_{S,1} & -8r_{S,2} \\ 0 & 16r_{S,2} \end{bmatrix}$$

On calcule ensuite les autres matrices supports.

$$\begin{aligned} W_{AB} = L_1.W_{SA} &= \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} r_{S,1} & -8r_{S,2} \\ 0 & 16r_{S,2} \end{bmatrix} = \begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 32r_{S,2} \end{bmatrix} \\ W_{BC} = L_2.W_{AB} &= \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 32r_{S,2} \end{bmatrix} = \begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 64r_{S,2} \end{bmatrix} \\ W_{CT} = M^{-1}.W_{BC} &= \begin{bmatrix} 1/2 & -1/6 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} 2r_{S,1} & 0 \\ 0 & 64r_{S,2} \end{bmatrix} = \begin{bmatrix} r_{S,1} & -32/3r_{S,2} \\ 0 & 64/3r_{S,2} \end{bmatrix} \end{aligned}$$

On peut maintenant vérifier la validité du rectangle de données.

$$\begin{aligned} |N(W_{CT})| &= \left| \frac{N(W_{BC})}{det(M)} \right| \\ |N(W_{CT})| &= \frac{64}{3} r_{S,1} r_{S,2} = 64 \\ \left| \frac{N(W_{BC})}{det(M)} \right| &= \frac{|128 r_{S,1} r_{S,2}|}{6} = 64 \end{aligned}$$

Notre choix était donc valide, l'application est enfin définie.

Remarque : Pour écrire cet exemple de modélisation d'applications en GMDSDF, j'ai dû refaire les calculs de Lee et Murthy. J'ai alors découvert que le calcul du déterminant de la matrice W_{CT} était faux. Cette erreur oblige Lee et Murthy à reconsidérer le rectangle de données du « décimateur » et à effectuer d'autres calculs qui sont donc inutiles.

1.4.1.4 L'application en ARRAY-OL

Contrairement à ce qui a été fait pour GMDSDF, la modélisation en ARRAY-OL ne nécessite pas l'ajout de tâches supplémentaires utilisées pour manipuler le treillis. Ainsi le modèle global comporte seulement trois tâches : les deux interpolations et le « décimateur ».

ARRAY-OL gère des tableaux et non des treillis de points, hors la réprésentation du treilis de l'application par un tableau implique la présence de case vide dans ce tableau. Or ces cases sont totalement artificielles, elles ne sont représentées ni dans le flux ni en mémoire, les données du tableau sont contiguës. Nous pouvons considérer la modélisation du flux comme un tableau dont la hauteur ne serait pas de 625 lignes mais de $\frac{625}{2}$ lignes.

La modélisation des modèles locaux est relativement simple et se base sur les informations présentes dans la section 1.4.1.2 page 32. Les données de la partie opérande de la première interpolation s'obtiennent par le raisonnement suivant :

- le tableau : la dimension temporelle est de taille infinie et la dimension verticale comporte 313 lignes puisque nous avons compacté le tableau ; finalement, le tableau est de taille $\binom{\infty}{313}$;
- le motif : la description de l'application nous donne un motif de taille $\binom{11}{5}$;
- l'origine : on commence l'interpolation dès la première ligne du signal, donc on a $\binom{0}{0}$ comme origine ;
- l'ajustage : la matrice d'ajustage est $\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ car tous les points du motif sont contiguës ;
- le pavage : la matrice de pavage est $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$ puisque le tableau a été compacté.

De façon similaire on construit le reste de la première interpolation et l'intégralité de la deuxième. Les résultats sont donnés dans le tableau 1.30.

	Tableau	Motif	Origine	Pavage	Ajustage
Interpolation 1 (opérande)	$\binom{\infty}{313}$	$\binom{11}{5}$	$\binom{0}{0}$	$\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$	$\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$
Interpolation 1 (résultat)	$\binom{\infty}{1250}$	(4)	$\binom{0}{0}$	$\left(\begin{smallmatrix}1&0\\0&4\end{smallmatrix}\right)$	$\binom{0}{1}$
Interpolation 2 (opérande)	$\binom{\infty}{1250}$	(3)	$\binom{0}{0}$	$\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$	$\binom{0}{1}$
Interpolation 2 (résultat)	$\binom{\infty}{2500}$	(2)	$\binom{0}{0}$	$\left(\begin{smallmatrix}1&0\\0&2\end{smallmatrix}\right)$	$\binom{0}{1}$

FIG. 1.30: Modélisation de l'application en ARRAY-OL

Seule la décimation est un plus complexe. Sa partie consommation prend un point sur trois dans chaque colonne, mais surtout elle effectue un décalage de trois points une fois sur deux pour pouvoir recréer l'interlacement (*cf* figure 1.27 page 33). La construction de la

matrice de pavage se fait selon le raisonnement suivant. Pour prendre les motifs verticalement, on utilise un simple vecteur $\binom{0}{6}$. Pour le déplacement horizontal le problème est plus difficile, car il faut effectuer le décalage une fois sur deux. Ainsi on utilise deux vecteurs de pavage supplémentaires $\binom{1}{3}$ et $\binom{2}{0}$ au lieu d'un seul. Le premier vecteur est utilisé pour effectuer le décalage en permettant un déplacement en diagonale. Le deuxième sert à se déplacer horizontalement (*cf* figure 1.31). Mais une telle modélisation impose que le premier vecteur ait une limitation d'itération de deux car sinon le décalage augmenterait d'une colonne à l'autre ; on passerait ainsi d'un décalage de trois à un décalage de six, etc. De plus les deux vecteurs entreraient en concurrence pour prendre des motifs dans les mêmes colonnes.

Cette limite de pavage est calculée à partir de la matrice de pavage de la partie résultat (*cf* figure 1.31). Or cette dernière qui comporte, elle aussi, trois vecteurs peut être construite sur le même principe. Le premier vecteur sert à produire les motifs verticalement, le deuxième à produire les motifs horizontalement en passant d'une colonne pair à une colonne impair, alors que le troisième vecteur permet de passer d'une colonne pair à l'autre. On obtient $P = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \end{pmatrix}$. Le calcul des bornes de pavage nous donne bien 2 comme limite du vecteur diagonale. En effet ARRAY-OL n'autorisant pas l'affectation multiple, le calcul des bornes limite les itérations sur le deuxième vecteur afin qu'il ne rentre pas « en concurrence » avec le troisième.

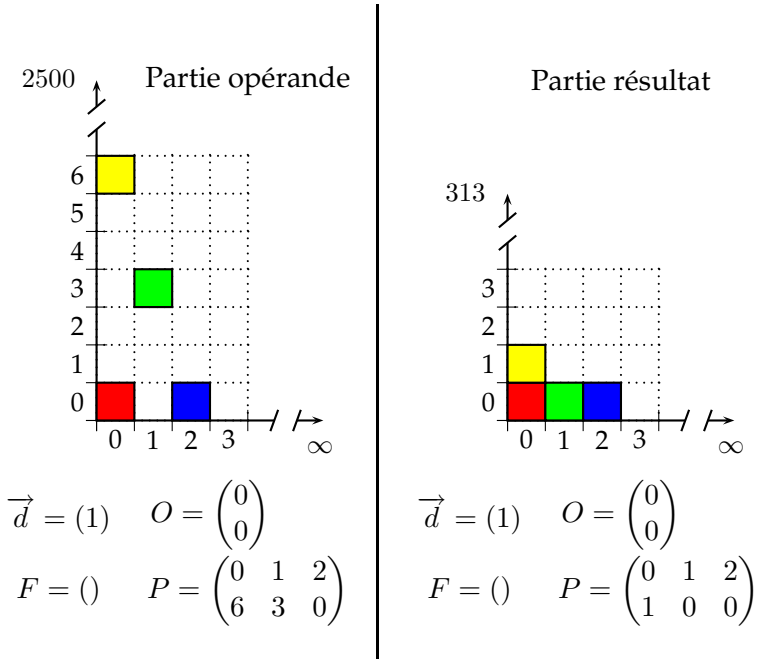


FIG. 1.31: Modélisation du « décimateur »

Cet exemple montre comment la modélisation en ARRAY-OL reste toujours très proche de l'application. Aucune tâche supplémentaire n'est ajoutée artificiellement pour gérer tel ou tel problème. L'ordonancement de l'application n'est pas abordé ici, nous verrons comment ordonnancer une application dans le chapitre 4 page 99.

1.4.2 Comparaison théorique

La comparaison théorique de GMDSDF et de ARRAY-OL doit se faire en deux parties. Tout d'abord, il faut comparer l'expressivité des modèles : peut-on représenter les mêmes applications en ARRAY-OL et en GMDSDF ? Puis il s'agit de comparer le calcul de l'ordonnancement, un ordonnancement en GMDSDF est il meilleur qu'en ARRAY-OL ?

1.4.2.1 Description des applications

Comment modéliser une application ? En premier lieu, rappelons quelles sont les étapes nécessaires à la modélisation d'une application en GMDSDF et en ARRAY-OL :

Pour GMDSDF :
- création du graphe ;
- choix des matrices exemples, d'« expansion » et de « décimation » pour les tâches spéciales ;
- choix des rectangles pour les tâches « ordinaires »
- calcul de l'ordonnancement ;
- calcul des matrices supports.

Pour ARRAY-OL :
- création du modèle global (qui n'est qu'un graphe) ;
- création des modèles locaux ;
- choix des matrices et des vecteurs pour chaque *tiler* ;
- calcul de l'ordonnancement.

Le nombre d'informations à donner est bien plus grand pour ARRAY-OL : tous les *tilers* doivent être complétés. Sachant qu'il y a approximativement deux *tilers* pour chaque arête du graphe et qu'il y a trois vecteurs et deux matrices à spécifier pour chaque *tiler*, on peut estimer le nombre d'informations à définir à dix fois le nombre d'arêtes. En revanche en GMDSDF, il n'y a que 2 rectangles à définir par arête plus une matrice pour chaque tâche spéciale. L'avantage va indéniablement à GMDSDF même si ARRAY-OL permet de spécifier toutes les données d'un *tiler* à l'aide d'une interface graphique. La difficulté de calcul des données n'est pas très élevée ni dans un cas ni dans l'autre.

Mais problème majeur de GMDSDF, le modeleur doit être capable de comprendre le calcul de l'ordonnancement et de le vérifier. Enfin, ARRAY-OL distingue clairement la manipulation de la structure des données (création, ajout ou suppression de dimensions) et le calcul sur les données elles mêmes. GMDSDF modélise les « décimateurs » au même niveau que les tâches dédiées aux calculs comme une FFT, alors qu'en ARRAY-OL tout est pris en charge par les *tilers*.

Quelles sont les limites d'une modélisation ? Bien que similaires, GMDSDF et ARRAY-OL n'autorisent pas la modélisation des mêmes applications.

La désignation des points à consommer ou à produire ne se fait pas de la même façon dans nos deux modèles. L'utilisation des matrices support pour GMDSDF permet de désigner des ensembles de points de forme plus irrégulière qu'en ARRAY-OL. Ainsi, l'exemple de la figure 1.32 n'est pas réalisable en ARRAY-OL. Nous ne sommes pas sûr que la désignation de tels ensembles de points représente un avantage quelconque pour GMDSDF. En effet, il est tout à fait possible de prendre une boite englobante pour effectuer la modélisation en ARRAY-OL et surtout il ne nous est jamais arrivé de rencontrer des tâches ayant besoin d'une

manipulation de données semblable. De plus en ARRAY-OL, contrairement à GMDSDF il est possible de désigner des ensembles disjoints de points comme sur le troisième exemple de la figure 1.20 page 25.

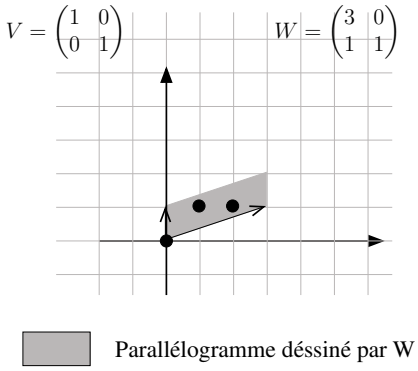


FIG. 1.32: Un exemple en GMDSDF irréalisable en ARRAY-OL

Un autre avantage d'ARRAY-OL sur GMDSDF est l'utilisation des tableaux toriques comme sur le cinquième exemple de la figure 1.20 page 25. Il ne s'agit pas d'un « gadget » mais d'une possiblité réellement intéressante, nous l'avons d'ailleurs utilisé dans plusieurs applications dont celle présentées dans **[**59**]** où une des dimensions toriques réprésente les sonars se trouvant autour d'un sous-marin et où une autre représente une dimension fréquentielle. GMDSDF permet d'utiliser les « états » et les « délais » ce qui n'est pas le cas d'ARRAY-OL. Pourtant ils sont indispensables à l'établissement de calculs simples comme la somme de vecteurs ; mais nous verrons que leur utilisation avec ARRAY-OL est difficile.

Mais la différence la plus notable entre GMDSDF et ARRAY-OL est sans nul doute que certaines des démonstrations servant de support aux principes de GMDSDF ne sont valides que pour des applications ayant aux plus deux dimensions. ARRAY-OL fonctionne avec des applications multidimensionnelles sans restrictions aucune sur le nombre maximum de ces dimensions. MDSDF, lui, n'est pas limité du point de vue des dimensions, en revanche il ne permet qu'une manipulation rudimentaire des données.

1.4.2.2 Calcul de l'ordonancement

Il est très difficile de comparer le calcul de l'ordonnancement d'ARRAY-OL et de GMDSDF. GMDSDF est un langage à flot de données basé sur SDF, ARRAY-OL est un simple langage de description. ARRAY-OL ne fournit donc aucune information sur son modèle d'exécution. Or l'ordonnancement doit s'adapter au modèle d'exécution.

En GMDSDF, calculer l'ordonnancement est une tâche difficile. Le problème principal vient des « décimateurs ». En effet il faut, pour ces derniers établir des contraintes dont la validation reste complexe. Cependant on obtient un ordonnancement permettant une exécution optimisée

En ARRAY-OL, l'utilisation du modèle d'exécution sous-jascent n'autorise qu'un ordonnancement basique où les tâches sont éxécutées séquentiellement : chacune attendant la fin de l'exécution de la précédente. Toutefois, l'exécution d'une tâche peut s'effectuer en bénéficiant du data-parallélisme du modèle. ARRAY-OL n'est donc pas aussi performant que

GMDSDF. Mais le chapitre 3 page 63 décrit comment il est possible de transformer une application ARRAY-OL afin d'optimiser son exécution.

1.4.3 Conclusion

ARRAY-OL et GMDSDF ont chacun leurs forces et leurs faiblesses. Ils ne sont d'ailleurs pas en mesure de décrire exactement les mêmes applications. Mais ARRAY-OL semble être le plus simple à manier malgré le nombre important d'informations qu'il nécessite. En ARRAY-OL, la manipulation de la structure des données et les calculs sur ces données ne s'effectuent pas au même niveau de modélisation et les phases de modélisation et d'ordonnancement sont clairement séparées. Mais surtout ARRAY-OL est le seul à ne souffrir d'aucune limite sur le nombre de dimensions ce qui fait de lui le seul modèle réellement multidimensionnel.

1.5 Conclusion

Dans ce chapitre, nous avons abordé la problématique de la modélisation des applications de traitement du signal systématique. Nous avons alors proposé le modèle flot de données pour effectuer cette modélisation. Puis nous avons présenté plusieurs langages basés sur ce modèle flot de données et nous avons vu qu'ARRAY-OL était particulièrement bien adapté. Enfin, nous avons étudié la modélisation concrète d'une application.

Notre motivation pour modéliser des applications de TSS était de réaliser les meilleures optimisations possibles, nous allons donc étudier l'optimisation des applications ARRAY-OL dans les chapitres suivants.

Chapitre 2

Optimisation des applications ARRAY-OL

我 看见 我 忘记。 我 听见 我 记住。 我 做 我 了解。

Proverbe chinois.

2.1 Introduction

Lorsque nous avons décrit ARRAY-OL dans le premier chapitre, nous avons vu qu'aucun modèle d'exécution ne lui était associé. Pourtant nous devons exécuter les applications que nous avons décrites, il faut donc que nous choisissions un modèle d'exécution. Mais un tel choix est source d'interrogations : quel est le modèle d'exécution le mieux adapté ? doit on limiter ARRAY-OL à un modèle d'exécution fixé comme pour SDF et ses dérivés ?

Les premiers travaux qui ont été menés sur l'exécution d'ARRAY-OL fournissent un ensemble de réponses intéressantes à ces questions. Cependant aucun de ces travaux ne parvienne à proposer une exécution optimale quel que soit le choix du modèle d'exécution. Mais ils montrent que modifier l'application en fonction du modèle d'exécution permet d'améliorer les résultats obtenus.

La première section de ce chapitre présente les travaux préliminaires qui ont été menés sur l'exécution des applications ARRAY-OL. La deuxième section aborde différentes méthodes pour modifier les applications. Enfin, dans la troisième section nous étudierons en détails le formalisme qui a été retenu : les ODT.

2.2 Etude de l'exécution d'ARRAY-OL

Les premiers travaux sur l'exécution d'une application ARRAY-OL ont été réalisés par Corinne Ancourt et François Irigoin dans le cadre d'une collaboration avec TUS **[**5**]** en 1997. Ces travaux étudient l'ordonnancement et le placement automatique d'une application sur une architecture SPMD.

Ils proposent pour cela d'utiliser un modèle concurrent de programmation logique par contrainte (CCLP) **[**23, 32**]**. Les contraintes sont établies à partir d'une formalisation de l'architecture, de l'application et du placement. Il est ainsi possible de tenir compte d'un grand

nombre de paramètres au cours de la formalisation : le nombre de processeurs, la taille de la mémoire, la latence maximum entre les entrées et les sorties... Les contraintes sont passées à un solveur qui utilise un ensemble d'heuristiques pour obtenir : un ordonnancement des calculs, un placement de ces calculs sur les processeurs et une gestion optimisée de la mémoire.

Au final, cette méthode est capable de générer automatiquement un ordonnancement séquentiel, pipeliné ou data-parallele, mais sans toutefois garantir que cet ordonnancement soit optimum. Afin d'améliorer la qualité des résultats, Ancourt et Iriguoin proposent de restructurer l'application pour obtenir un plus haut degré de parallélisme et pour améliorer la localité des données. Les outils de transformations d'applications ARRAY-OL décrits dans cette thèse pourraient donc être utilisés en aval de ces calculs d'ordonnancement.

2.3 Etude de l'optimisation d'ARRAY-OL

Les travaux sur l'exécution des applications ARRAY-OL sont directement issus des résultats de l'étude précédente et sont à l'origine de la collaboration entre TUS et le LIFL. TUS avait également pour motivation, en créant ce partenariat, de fournir aux concepteurs d'applications un outil d'optimisation qui reste le plus compréhensible possible. En effet la transformation d'applications permet aux concepteurs de voir les optimisations tout en ayant une compréhension parfaite puisque l'application reste exprimée en ARRAY-OL.

Ancourt et Iriguoin donnent plusieurs pistes [11, 43, 57] pour transformer une application ; ces pistes reposent toutes sur l'utilisation des transformations de boucles. En effet il est possible de représenter facilement une application ARRAY-OL par une succession de nids de boucles. Cependant nous verrons que cette solution n'a pas été retenue, c'est un formalisme conçu spécialement pour ARRAY-OL par TUS qui a été choisi. Toutefois, afin de mieux comprendre les enjeux et les difficultés de la transformation d'une application ARRAY-OL, je présente ci-dessous une analyse de l'utilisation des transformations de boucles sur ARRAY-OL.

2.3.1 ARRAY-OL sous forme de nids de boucles

Pour représenter une application ARRAY-OL en nids de boucles nous allons d'abord étudier la représentation d'un modèle local puis nous verrons comment enchaîner ces représentations pour former un modèle global.

2.3.1.1 Le modèle local

De façon schématique, un modèle local se déroule de la manière suivante : à chaque itération de pavage une tâche consomme des données dans un tableau d'entrée, les range dans un motif opérande, utilise ce motif pour calculer un motif résultat et range les éléments de ce dernier dans un tableaux de sortie. Les liens entre les éléments d'un motif et les éléments de son tableau associé sont donnés par la notion d'ajustage. On peut modéliser ce fonctionnement sous la forme de la formule (*cf* section 1.3.2.2 page 23) :

$$\forall \overrightarrow{x}, \overrightarrow{0} \leq \overrightarrow{x} < \overrightarrow{Q}, \forall \overrightarrow{y}, \overrightarrow{0} \leq \overrightarrow{y} < \overrightarrow{D}, (\overrightarrow{O} + P \times \overrightarrow{x} + F \times \overrightarrow{y}) \mod \overrightarrow{m}$$

Pour mémoire nous rappelons que $\overrightarrow{x}$ représente l'itération de pavage courante et $\overrightarrow{y}$ l'itération d'ajustage courante. On peut déduire, à l'aide de cette équation, une représentation

sous forme de nids de boucles de la construction d'un motif pour une itération de pavage fixée :

```
Pour y_1 Allant_de 0 A D_1 Faire
  ...
    Pour y_n Allant_de 0 A D_n Faire
      i→ = (O→ + P × x→ + F × (y_1 ... y_n)) mod m→;
      Motif[y_1, .., y_n] = Tableau[i→];
    FinPour
  ...
FinPour
```

Algorithme 2.1: Construction d'un motif

Puis en utilisant la description schématique du modèle local, on peut facilement extrapoler sa représentation sous forme de nids de boucles :

```
1  Pour x_1 Allant_de 0 A Q_1 Faire
2    ...
3      Pour x_f Allant_de 0 A Q_f Faire
4        Pour y_op,1 Allant_de 0 A D_op,1 Faire
5          ...
6            Pour y_op,n Allant_de 0 A D_op,n Faire
7              i→ = (O_op→ + P_op × (x_1 ... x_f) + F_op × (y_op,1 ... y_op,n)) mod m_op→
8              MotifOpérande[y_op,1, .., y_op,n] = TableauDEntrée[i→];
9            FinPour
10         ...
11       FinPour
12       Motif_Résultat=Traitement(Motif_Opérande);
13       Pour y_res,1 Allant_de 0 A D_res,1 Faire
14         ...
15           Pour y_res,n Allant_de 0 A D_res,n Faire
16             j→ = (O_res→ + P_res × (x_1 ... x_f) + F_res × (y_res,1 ... y_res,n)) mod m_res→
17             TableauDeSortie[j→] = MotifRésultat[y_res,1, .., y_res,n];
18           FinPour
19         ...
20       FinPour
21     FinPour
22   ...
23 FinPour
```

Algorithme 2.2: Un modèle local complet

Ce nid de boucles peut être décomposé en quatre parties distinctes :

1. les boucles pour représenter les itérations de pavage qui sont globales à la tâche : des lignes n° 1 à n° 3 et des lignes n° 21 à n° 23.
2. les boucles pour représenter les itérations d'ajustage opérande pour construire le motif opérande : des lignes n° 4 à n° 11.
3. le traitement du motif : ligne n° 12
4. les boucles pour représenter les itérations d'ajustage résultat pour produire le tableau de sortie : des lignes n° 13 à n° 20.

Cette représentation n'est toutefois pas complète car nous n'avons pris en compte que les tâches n'ayant qu'un seul tableau opérande et qu'un seul tableau résultat. Dans le cas d'une tâche à tableaux multiples, il faudrait ajouter la représentation de l'ajustage de chacun des autres tableaux. Ainsi, pour un tableau opérande l'insertion se ferait entre les lignes n° 11 et n° 12 et pour un tableau résultat entre les lignes n° 20 et n° 21.

Mais cela n'a que peu d'importance car le traitement qu'effectue la tâche (ligne n° 12) ne peut être modifié. En effet les calculs réalisés par ce traitement consomment et produisent toujours des motifs identiques. On peut donc se limiter à une représentation du modèle local sans ajustage dans l'optique d'une manipulation par transformations de boucles. On obtient ainsi la représentation généralisée suivante :

```
Pour x_1 Allant_de 0 A Q_1 Faire
  ...
    Pour x_f Allant_de 0 A Q_f Faire
      i⃗ = (O_op⃗ + P_op × (x_1 ... x_f)) mod m_op⃗
      MotifOpérande[...] = TableauDEntrée[i⃗];
      Motif_Résultat=Traitement(Motif_Opérande);
      j⃗ = (O_res⃗ + P_res × (x_1 ... x_f)) mod m_res⃗
      TableauDeSortie[j⃗] = MotifRésultat[...];
    FinPour
  ...
FinPour
```

Algorithme 2.3: Un modèle local pour les transformations de boucle

2.3.1.2 Le modèle global

Ce dernier est simplement représenté comme une suite de modèles locaux, on obtient donc une succession de nids de boucles pour représenter une tâche ARRAY-OL.

2.3.1.3 Les tâches hiérarchiques

Lorsqu'une tâche est hiérarchique, on se contente de remplacer le traitement du motif par la représentation en nids de boucles du niveau inférieur de la hiérarchie.

2.3.1.4 Spécificités des boucles ARRAY-OL

Les particularismes d'ARRAY-OL impliquent de fait un certain nombre de spécifités dans les nids de boucles :

- deux modèles locaux qui se suivent n'ont pas nécessairement des bornes de boucles identiques ;
- le nombre de boucles peut même varier d'un modèle local à l'autre ;
- les boucles sont parfaitement imbriquées donc leur ordre n'a aucune importance ;
- l'accès aux tableaux n'est pas linéaire à cause de la présence d'un modulo.

2.3.1.5 Des nids de boucles à ARRAY-OL

Nous allons transformer dans la section suivante des nids de boucles, or il faut que le résultat de cette transformation corresponde toujours à une application ARRAY-OL. Pour

distinguer les nids de boucles correctes, nous donnons ci-dessous une liste d'impératifs qui doivent être respectés :

- les boucles sont parfaitement imbriquées ;
- aucune instruction ne se situe entre les boucles ;
- une boucle varie entre zéro et une constante ;
- l'incrément de la boucle est de un ;
- l'accès au tableau se fait par une expression de ce type : $(\overrightarrow{O} + P \times \overrightarrow{x} + F \times \overrightarrow{y}) \mod \overrightarrow{m}$.

2.3.2 Les transformations de boucles

Nous venons de voir qu'il est possible de modéliser une application ARRAY-OL à l'aide de nids de boucles. Étudions maintenant les transformations de boucles que nous pouvons utiliser.

2.3.2.1 Intérêt des transformations

Les transformations de boucles sont le plus souvent utilisées dans les compilateurs afin d'améliorer l'exécution des applications compilées. Il y a de nombreuses optimisations possibles, on peut toutefois en distinguer quelques une :

1. l'augmentation du degré de parallélisme qui consiste à placer un maximum d'instructions dans des boucles parallèles **[**2**]** ; il existe plusieurs variantes qui demandent de minimiser ou de maximiser le nombre final de boucles ;
2. la minimisation des barrières de synchronisation qui consiste à rendre indépendante une suite de boucles afin de pouvoir les exécuter parallèlement **[**11**]** ;
3. l'amélioration de la localité des données qui consiste à rapprocher des instructions manipulant des données identiques. L'optimisation, dans ce cas, se situe non pas au niveau de l'amélioration du parallélisme mais au niveau de la réduction des accès mémoires, ce qui a pour effet d'augmenter la vitesse d'exécution. Il est également possible de diminuer dans certains cas la quantité de mémoire nécessaire à l'exécution en réduisant la taille des tableaux manipulés par les boucles **[**36**]**.

Pour se faire une idée plus précise des enjeux et de la difficulté d'utilisation de ces optimisations, le lecteur pourra se référer à l'article **[**18**]**.

Il nous faut maintenant choisir quelle optimisation appliquer dans le cadre d'ARRAY-OL. Notre objectif principal n'est pas d'améliorer le parallélisme de l'application puisque les nids de boucles des modèles locaux sont déjà tous parallèles. En revanche, il semble pertinent d'augmenter la localité des données. En effet, les tableaux produits par un modèle local, donc par un nid de boucles, sont consommés par le modèle local suivant, regrouper ces tableaux au sein d'un même nid de boucles serait donc avantageux et permettrait de réduire éventuellement leurs tailles. De plus cela réduirait l'impact des barrières de synchronisation se trouvant entre chaque modèle local[1].

On trouve dans la littérature de nombreux travaux sur l'amélioration de la localité des données et la réduction de l'occupation mémoire. On peut citer notamment :

[1] Les tableaux produits par un modèle local étant consommés par le modèle local suivant, il y a nécessairement une barrière de synchronisation entre les deux

- Manjikian et Abdelrahman [45] qui proposent un algorithme permettant la fusion de boucles par utilisation des décalages d'itérations sur des applications à dépendances de données uniformes ;
- McKinley et Kennedy [36, 47, 13] qui proposent d'augmenter la localité des données, mais sans possibilités de réductions de la taille des tableaux ;
- Gao et Sarkar [28, 27] qui proposent de maximiser le nombre de tableaux à réduire, mais qui imposent l'assignation unique des données dans les tableaux, une seule utilisation des indices de boucles dans l'accès aux tableaux et qui limitent à deux la profondeur du nid de boucles ;
- Fraboulet [26] qui propose un algorithme exponentiel offrant une résolution optimale de la réduction de la taille des tableaux.

Mais la mise en œuvre de ces différentes techniques, bien que conservant la sémantique des applications, en modifie la syntaxe. En effet, la forme des boucles et l'emplacement des instructions ne respectent aucun schéma particulier. Or comme nous l'avons vu ci-dessus, il faut que le résultat de la transformation d'une application ARRAY-OL corresponde toujours à une application ARRAY-OL. Nous nous retrouvons donc dans un cas original qui n'a pas été traité.

2.3.2.2 Quelques transformations de boucles

La liste que nous donnons ci-dessous n'est pas exhaustive, nous nous limitons aux transformations qui peuvent nous servir. Toutefois une taxonomie détaillée est disponible dans [37, 6, 29].

Fusion de boucles : La fusion de boucles permet de regrouper deux boucles consécutives en une seule. Ces deux boucles doivent avoir le même nombre d'itérations et des bornes de boucles identiques. Mais le résultat d'une fusion n'est valide que si les dépendances de données n'ont pas été modfiées. Nous reparlerons de ces dépendances de données dans la section 2.3.2.4 page 49.

```
Pour i Allant_de 0 A N Faire
  a[i]=...
FinPour
Pour i Allant_de 0 A N Faire
  ...=a[i]
FinPour
```

avant la fusion

```
Pour i Allant_de 0 A N Faire
  a[i]=...
  ...=a[i]
FinPour
```

après la fusion

L'intérêt de la fusion de boucles est évident pour augmenter la localité des données. Ainsi dans l'exemple ci-dessus, la production et la consommation d'une case du tableau `a` deviennent concomitantes. Il est même possible de réduire la taille des tableaux manipulés. Dans notre exemple, si le tableau `a` n'est plus utilisé dans la suite de l'application, on peut le remplacer par un scalaire. De nombreux travaux traitent de la réduction en taille des tableaux [12, 16, 52, 53] à l'aide de la fusion.

L'utilisation de la fusion de boucle dans ARRAY-OL est difficile puisque comme nous l'avons vu les bornes des boucles ne sont pas identiques d'un nid à l'autre. Il nous faut donc harmoniser les bornes de boucles entre les nids avant de pouvoir fusionner.

Permutation de boucles : La permutation de boucles [12, 47] consiste à échanger l'ordre de deux boucles consécutives, ceci n'est possible que si les boucles peuvent s'exécuter de manière parallèle.

```
Pour i Allant_de 0 A N Faire
  Pour j Allant_de 0 A M Faire
    S1
    S2
  FinPour
FinPour
```

avant la permutation

```
Pour j Allant_de 0 A M Faire
  Pour i Allant_de 0 A N Faire
    S1
    S2
  FinPour
FinPour
```

après la permutation

Un modèle local ARRAY-OL étant data-parallèle, il est possible d'échanger l'ordre des boucles sans aucune limitations.

Pavage de boucles : Le pavage de boucles transforme une boucle en deux boucles imbriquées, pour cela il divise l'espace d'itération de cette boucle d'un facteur `K`. La borne d'itération de la boucle doit être divisible par `K` pour ne pas contraindre à l'ajout d'une fonction minimum dans le calcul de la borne de la nouvelle boucle.

```
Pour i Allant_de 0 A N Faire
  a[i]=...;
  b[i+2]=a[i];
FinPour
```

avant le pavage de boucles

```
Pour i Allant_de 0 A N/K Faire
  Pour j Allant_de 0 A K Faire
    a[i*P+j]=...;
    b[i*P+j+2]=a[i*P+j];
  FinPour
FinPour
```

après le pavage de boucles

Nous avons vu que pour effectuer une fusion de boucles nous avions besoin de boucles ayant des bornes identiques. Or une utilisation possible du pavage de boucles est justement de rendre deux bornes identiques : soit `N1` et `N2` deux bornes de boucles, pour rendre ces boucles fusionables il suffit de choisir `K1` et `K2` tel que `N1/K1=N2/K2`.

2.3.2.3 Vers une technique d'optimisation

La combinaison de la fusion, du pavage et de la permutation de boucles peut donc nous permettre d'améliorer la localité des données d'une application ARRAY-OL. Notre optimisation se décompose en quatre étapes :

1. On a deux modèles locaux consécutifs représentant ainsi deux tâches.
2. Pour autoriser la fusion, on choisit une boucle dans chaque modèle local et on effectue un pavage de boucles. Le choix des deux boucles est pour l'instant fait au hasard, mais nous verrons en fait qu'il est crucial.
3. On utilise la permutation pour mettre nos deux boucles au même niveau.
4. On fusionne. On peut être amené à effectuer plusieurs fusions : si `b` est produit élément par élément à l'aide de deux boucles et s'il est consommé de la même façon, il est alors intéressant de faire deux fusions successives afin d'augmenter encore la localité des données.

Le résultat obtenu semble ne pas correspondre à une application ARRAY-OL puisqu'on n'obtient pas un nid de boucles parfaitement imbriquées. Mais ce n'est pas le cas. En effet, nous avons obtenu une tâche hiérarchique : la boucle d'indice `j` représente le niveau supérieur, les

boucles d'indices i et k la première sous-tâche et la boucle d'indice l la deuxième sous-tâche. Nous avons donc fusionné nos deux tâches en une seule, mais en créant une hiérarchie.

```
Pour i Allant_de 0 A N1 Faire
  Pour j Allant_de 0 A M1 Faire
    ...=a[...]
    TE1
    b[...]=...
  FinPour
FinPour
Pour l Allant_de 0 A N2 Faire
  ...=b[...]
  TE2
  c[...]=...
FinPour
```

étape n° 1

```
Pour i Allant_de 0 A N1 Faire
  Pour j Allant_de 0 A M1/K1 Faire
    Pour k Allant_de 0 A K1 Faire
      ...=a[...]
      TE1
      b[...]=...
    FinPour
  FinPour
FinPour
Pour l Allant_de 0 A N2/K2 Faire
  Pour m Allant_de 0 A K2 Faire
    ...=b[...]
    TE2
    c[...]=...
  FinPour
FinPour
```

étape n° 2

```
Pour j Allant_de 0 A M1/K1 Faire
  Pour i Allant_de 0 A N1 Faire
    Pour k Allant_de 0 A K1 Faire
      ...=a[...]
      TE1
      b[...]=...
    FinPour
  FinPour
FinPour
Pour l Allant_de 0 A N2/K2 Faire
  Pour m Allant_de 0 A K2 Faire
    ...=b[...]
    TE2
    c[...]=...
  FinPour
FinPour
```

étape n° 3

```
Pour j Allant_de 0 A M1/K1 Faire
  Pour i Allant_de 0 A N1 Faire
    Pour k Allant_de 0 A K1 Faire
      ...=a[...]
      TE1
      b[...]=...
    FinPour
  FinPour
  Pour m Allant_de 0 A K2 Faire
    ...=b[...]
    TE2
    c[...]=...
  FinPour
FinPour
```

étape n° 4

Nous avons réussi à fusionner deux modèles locaux ARRAY-OL tout en restant dans le domaine des applications ARRAY-OL. Cependant, nous devons nous poser deux questions :

1. Le résultat obtenu est il correct ? : comme nous l'avons vu ci-dessus il faut que les dépendances de données soient toujours les mêmes. Dans le cadre d'ARRAY-OL, il faut que les données produites dans le tableau b au niveau de la première sous-tâche soient suffisantes pour pouvoir exécuter TE2 dans la deuxième sous-tâche. Le choix des boucles à fusionner est donc capital pour obtenir un résultat correct.
2. Avons nous réellement optimisé notre application ? : l'amélioration la plus importante se situe au niveau du tableau b, à l'image de l'exemple sur la fusion de boucle (*cf* section 2.3.2.2 page 46), il est possible de réduire la taille de b puisqu'il n'est ni produit ni consommé de manière complète. L'augmentation de la localité des données est double. Premièrement la consommation et la production de b sont séparées par moins de boucles : dans notre exemple la boucle d'indice j n'est plus présente. Deuxième-

ment la réduction de la taille de b permet de rendre b plus facilement accessible dans les hiérarchies mémoires

Notre technique d'optimisation semble donc valable, mais encore faut-il que nous trouvions un moyen de vérifier l'exactitude de son application et que nous soyons capables de réduire la taille de b.

2.3.2.4 Dépendances de données

Pour appliquer les optimisations, il nous faut donc connaître les dépendances de données présentes dans l'application. En effet, la cohésion des dépendances est nécessaire pour qu'une transformation ne modifie pas les résultats d'une application. Il existe plusieurs modèles pour représenter et manipuler les dépendances de données, Mais, l'utilisation de tableaux toriques dans ARRAY-OL complexifie grandement la manipulation des dépendances car elle les rend non linéaires. Nous ne détaillerons donc pas ici les techniques habituelles, toutefois le lecteur trouvera des études très détaillées dans **[**25, 17**]**. La manipulation des dépendances est le réel défi de la transformation d'applications ARRAY-OL. C'est pour cette raison que TUS a proposé son propre formalisme de manipulations de dépendances : les ODT.

2.4 Formalisme ODT

Le formalisme ODT (Opérateurs de Description de Tableau) a été inventé par Alain Demeure (*cf* section 1.3.1.2 page 19 et **[**20**]**) afin d'exprimer les dépendances entre les parties opérandes et résultats d'une tâche ARRAY-OL. Il est basé sur l'algèbre linéaire avec contraintes et est constitué de plusieurs opérateurs définissant les liens entre deux espaces $\mathbb{K}^n$. On peut relier ces opérateurs donc ces espaces à l'aide d'une loi de composition. Pour transcrire en ODT les dépendances d'une tâche ARRAY-OL, il faut identifier les points des tableaux à leurs coordonnées. Puis c'est sur ces coordonnées que sont appliqués les opérateurs. On obtient ainsi une suite d'opérateurs exprimant les dépendances. Il est alors possible d'effectuer un certain nombre de calculs découlant des propriétés intrinsèques à ARRAY-OL. Nous allons donc étudier le formalisme ODT et ses opérateurs, puis la représentation d'une tâche ARRAY-OL et enfin nous décrirons quelques propriétés mathématiques de cette représentation.

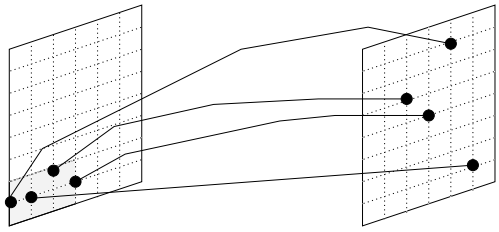
2.4.1 Présentation des ODT

2.4.1.1 Opérateurs élémentaires

Chaque opérateur est en fait une relation binaire de $\mathbb{K}^n$ dans $\mathbb{K}^m$ et se comprend par une lecture de la droite vers la gauche. Dans le contexte particulier d'ARRAY-OL, nous utiliserons le plus souvent $\mathbb{Z}$ comme espace car les opérateurs sont appliqués aux coordonnées de points d'un tableau. Cependant, nous verrons que parfois nous sommes obligés d'utiliser $\mathbb{Q}$. Au total, il existe neuf opérateurs.

Modulo : M

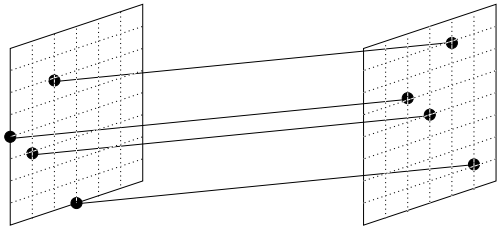
$$\forall \overrightarrow{m} \in (\mathbb{Z}^*)^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \overrightarrow{y}\, M\, \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} \mod \overrightarrow{m}$$



Une valeur infinie dans le modulo indique que seules les valeurs positives sont gardées. Le modulo se note $(\overrightarrow{m})_{\mathbf{M}}$

Shift : S

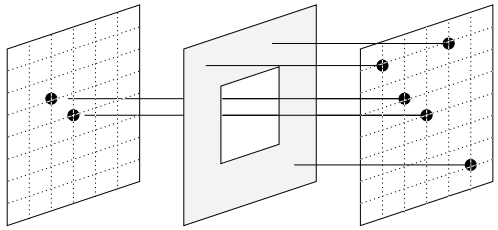
$$\forall \overrightarrow{shift} \in \mathbb{K}^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \overrightarrow{y} S \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{shift}$$



Il s'agit simplement d'une opération de translation. Elle se note $\left(\overrightarrow{shift}\right)_{\mathbf{S}}$.

Gabarit : G

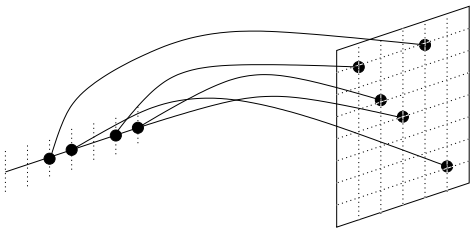
$$\forall (\overrightarrow{min}, \overrightarrow{max}) \in \mathbb{Z}^n \times \mathbb{Z}^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \overrightarrow{y} G \overrightarrow{x} \Leftrightarrow \overrightarrow{min} \leq \overrightarrow{x} < \overrightarrow{max} \text{ et } \overrightarrow{x} = \overrightarrow{y}$$



Le gabarit agit comme un filtre laissant passer certains points et en bloquant d'autres. Il se note $\left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}$ ou simplement $(\overrightarrow{max})_{\mathbf{G}}$ si $\overrightarrow{min}$ est nul. En outre une valeur infinie indique une absence de contrainte.

Projection : Pro

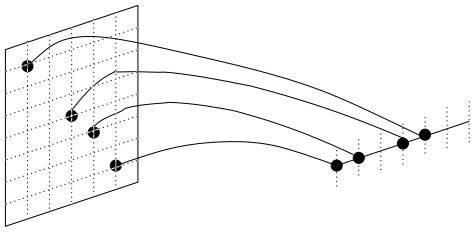
$$\forall \mathcal{M} \in M_{mn}(K), \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^m, \overrightarrow{y} \text{ Pro } \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \mathcal{M}.\overrightarrow{x}$$



La projection est une multiplication matricielle et permet donc d'effectuer des changements de repères ou de calculer le résultat d'applications linéaires. Elle se note $\left|\mathcal{M}\right|$. La taille de l'espace de départ est égale aux nombre de colonnes de $\mathcal{M}$ alors que le nombre de lignes est égale à la taille de l'espace d'arrivée.

Segmentation : Seg

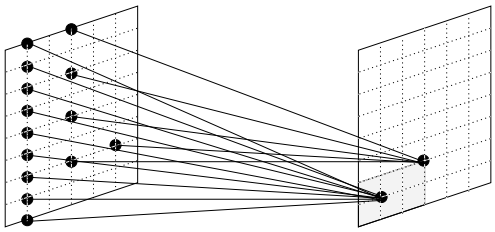
$$\forall \mathcal{M} \in M_{mn}(K), \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^m, \overrightarrow{y} \text{ Seg } \overrightarrow{x} \Leftrightarrow \overrightarrow{x} = \mathcal{M}.\overrightarrow{y}$$



La matrice $\mathcal{M}$ d'une projection n'étant pas forcement inversible la segmentation sert de notation pour exprimer une « inversion théorique ». Elle se note $\overline{\mathcal{M}}$. La concordance entre le nombre de lignes et de colonnes avec la dimension des espaces est inversée par rapport à la projection.

Éclatement : E

$$\forall \overrightarrow{m} \in (\mathbb{Z}^*)^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \overrightarrow{0} \leq \overrightarrow{x} < \overrightarrow{m},$$
$$\overrightarrow{y} E \overrightarrow{x} \Leftrightarrow \forall \overrightarrow{k} \in \mathbb{K}^n, \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{k} \times \overrightarrow{m}$$



Une valeur infinie dans l'éclatement indique que seules les valeurs positives sont gardées. L'éclatement se note $(\overrightarrow{m})_\star$.

Arrondi : A

$$\forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{Q}^n \times \mathbb{Z}^n, \overrightarrow{y} A \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \lfloor \overrightarrow{x} \rfloor$$

L'arrondi se note $\lfloor$.

Fractionneur : F

$$\forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{Z}^n \times \mathbb{Q}^n, \exists \overrightarrow{r} \in \mathbb{Q}^n, \overrightarrow{0} \leq \overrightarrow{r} < \overrightarrow{1}, \overrightarrow{y} F \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{r}$$

Le fractionneur se note $\rfloor$.

2.4.1.2 Opérations élémentaires

Application d'un ODT On note

$$O \blacktriangleleft \overrightarrow{x} = \{\overrightarrow{y} \mid \overrightarrow{y} O \overrightarrow{x}\}$$

l'application de l'ODT O au vecteur $\overrightarrow{x}$.

Loi de composition Il existe une loi de composition sur les ODT, elle est identique à celle des relations. Elle ne permet donc de composer que des ODT qui ont des espaces d'arrivée et de départ communs. Elle se lit de droite à gauche et se note « . ».

ODT miroirs Dans la suite, nous avons besoin de rendre symétriques 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

$$(ODT_1.ODT_2)^{-1} = ODT_2^{-1}.ODT_1^{-1}$$

Il nous suffit donc de décrire les miroirs des opérateurs élémentaires :

Gabarit $\left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}{}^{-1} = \left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}$

Décalage $\left(\overrightarrow{shift}\right)_{\mathbf{S}}{}^{-1} = \left(-\overrightarrow{shift}\right)_{\mathbf{S}}$

Modulo $\left(\overrightarrow{m}\right)_{\mathbf{M}}{}^{-1} = \left(\overrightarrow{m}\right)_{\star}$

Éclatement $\left(\overrightarrow{m}\right)_{\star}{}^{-1} = \left(\overrightarrow{m}\right)_{\mathbf{M}}$

Projection $\left|\mathcal{M}\right|^{-1} = \underline{\mathcal{M}}$

Segmentation $\underline{\mathcal{M}}^{-1} = \left|\mathcal{M}\right|$

Arrondi $\lfloor^{-1} = \rfloor$

Fractionneur $\rfloor^{-1} = \lfloor$

Attention, la symétrie n'est pas, à proprement parler, 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.

2.4.2 Liens avec ARRAY-OL

2.4.2.1 Représentation d'une demi-tâche

Avant de pouvoir exprimer les dépendances qui unissent les parties opérandes et résultats d'une tâche ARRAY-OL, il faut déterminer quels liens nous voulons représenter et surtout dans quels espaces nous allons nous situer.

Dans une configuration comme celle d'une tâche ARRAY-OL, la principale question sur les dépendances est : « qui produit quoi ? ». Il n'est pas possible de savoir que tel point sert à produire tel autre point. En effet ce genre de dépendance est « noyé » dans les motifs, on sait juste que tel groupe de motifs produit tel autre groupe. En d'autres termes à partir d'un motif résultat on peut retrouver les motifs opérandes qui ont servi à le produire et réciproquement, c'est une des conséquences de l'assignation unique.

La seule méthode pour désigner en ARRAY-OL un motif particulier dans une tâche est de donner la valeur des itérations de pavage (*cf* encadré page 53), ce qui désigne par là même les autres motifs produits et consommés à cette itération. On se servira alors de l'« espace Q » (ou espace de pavage) comme intermédiaire entre les parties opérandes et résultats d'une tâche.

En revanche, il nous faut également conserver un lien entre les tâches elles-mêmes afin de pouvoir formaliser le déroulement de l'application. Or seuls les tableaux assurent ce rôle : une tâche est reliée à une autre tâche uniquement par leur(s) tableau(x) commun(s). Il faut donc trouver un ensemble servant de représentation aux tableaux qui permette de passer à l'espace d'itération. Rien n'est plus simple, il suffit de prendre l'espace des coordonnées des points qui a autant de dimensions que le tableau représenté.

En résumé pour modéliser une tâche, il faut passer de l'ensemble des coordonnées des éléments du tableau opérande à l'espace d'itération Q. On peut l'associer à « l'espace D » (ou espace d'ajustage) afin de conserver le lien entre les points et les motifs. On obtient ainsi l'espace QD, qui est constitué de toutes les valeurs de pavage et d'ajustage. Puis à partir de cet espace et grâce aux valeurs de pavage, il est possible de faire la transition avec la

Espace d'itération

En ARRAY-OL, les itérations sur les vecteurs de la matrice de pavage permettent de connaître les premiers éléments des motifs. À l'aide des valeurs d'itérations, on peut donc désigner de manière unique chaque motif et comme une itération de pavage marque une exécution de la tâche, on désigne également tous les motifs consommés et produit à cette exécution. On nomme « espace Q », l'espace constitué des valeurs prises lors des différentes itérations, cet espace a autant de dimensions qu'il y a de vecteurs dans une matrice de pavage, puisque chaque valeur d'itération est couplée à un vecteur, et les dimensions sont égales aux bornes de pavages. Symétriquement, on désigne par « espace D », l'espace constitué des valeurs prises par les différentes itérations d'ajustages. Cet espace est construit de la même façon mais il n'est pas global à la tâche, il en existe un pour chaque demi-tâche. L'espace d'ajustage a autant de dimensions qu'il y a de vecteurs d'ajustage donc que de dimensions au motif.

partie résultat où on applique le cheminement inverse pour retomber sur les coordonnées des points du tableau résultat, à partir desquels on peut enchaîner la représentation ODT de la tâche suivante.

Voyons maintenant comment formaliser ce raisonnement avec les ODT. Dans un premier temps nous étudions le passage entre l'espace QD et l'espace des coordonnées. Considérons d'abord l'espace QD, filtrons l'ensemble des points avec un gabarit qui a comme valeur de $\overrightarrow{max}$ les bornes de pavage et d'ajustage. Nous sommes désormais sûrs de ne traiter que des éléments appartenant à l'espace d'itération de la tâche. Il faut ensuite arriver à exprimer avec des ODT la formule 1.8 page 25 ($\forall \overrightarrow{X_d}, \overrightarrow{0} \leq \overrightarrow{X_d} < \overrightarrow{D}, (\overrightarrow{O} + P \times \overrightarrow{X_q} + F \times \overrightarrow{X_d}) \mod \overrightarrow{m}$) présentée qui exprime les liens reliant l'espace QD et l'espace des coordonnées. On réalise la partie linéaire en utilisant une projection avec comme valeur de (M) la juxtaposition de P et F puis en utilisant le *shift* pour le décalage à l'origine. Enfin, on utilise le modulo pour le modulo correspondant de la partie non linéaire. On arrive à :

$$(M)_{\mathbf{M}} . (O)_{\mathbf{S}} . \left| P \quad F \right| . \begin{pmatrix} Q \\ D \end{pmatrix}_{\mathbf{G}}$$

Ainsi, on obtient le lien entre les éléments de ces deux ensembles et par là même la représentation d'une demi-tâche. Désormais, il nous suffit de représenter une tâche complète.

2.4.2.2 Représentation d'une tâche complète

Nous avons deux expressions représentant respectivement la partie opérande et résultat de la tâche.

$$(M_{op})_{\mathbf{M}} . (O_{op})_{\mathbf{S}} . \left| P_{op} \quad F_{op} \right| . \begin{pmatrix} Q \\ D_{op} \end{pmatrix}_{\mathbf{G}}$$

$$(M_{res})_{\mathbf{M}} . (O_{res})_{\mathbf{S}} . \left| P_{res} \quad F_{res} \right| . \begin{pmatrix} Q \\ D_{res} \end{pmatrix}_{\mathbf{G}}$$

Mais les gabarits, donc les espaces, ne sont pas identiques. En effet, les itérations d'ajustage ne concordent pas, ce qui est logique puisque les motifs opérandes et résultats sont définis indépendamment les uns des autres. Seul l'espace d'itération de pavage correspond. Afin d'obtenir facilement un enchaînement, on agrandit les espaces de chaque côté en ajoutant l'espace d'ajustage se trouvant « en face ».

Mais ce changement d'espace entraîne une modification de l'expression ODT. Le gabarit devient $\begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}$ et ce changement va se propager et risque s'il n'est pas endigué d'amener des résultats faux. Donc au niveau de la projection on fait correspondre chaque vecteur d'ajustage à 0, ce qui supprimera leur incidence sur les calculs, on passe de $P \times X_q + F \times X_{d_{op}}$ à $P \times X_q + F \times X_{d_{op}} + 0 \times X_{d_{res}}$, ce qui ne change rien au résultat et on obtient :

$$\left(M_{op}\right)_{\mathbf{M}} . \left(O_{op}\right)_{\mathbf{S}} . \left|P_{op} \quad F_{op} \quad 0\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}}$$

$$\left(M_{res}\right)_{\mathbf{M}} . \left(O_{res}\right)_{\mathbf{S}} . \left|P_{res} \quad 0 \quad F_{res}\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}}$$

Pour les concaténer afin d'obtenir une expression complète, nous allons inverser la deuxième expression à l'aide des ODT miroirs. On obtient :

$$\begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad F_{res}}} . \left(-O_{res}\right)_{\mathbf{S}} . \left(M_{res}\right)_{\star}$$

Ce qui finalement donne :

$$\left(M_{op}\right)_{\mathbf{M}} . \left(O_{op}\right)_{\mathbf{S}} . \left|P_{op} \quad F_{op} \quad 0\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad F_{res}}} . \left(-O_{res}\right)_{\mathbf{S}} . \left(M_{res}\right)_{\star}$$

Mais comme nous l'avons vu dans la section 1.3.2.3 page 26 nous savons que la demi-tâche résultat est nécessairement « exacte ». Ainsi elle doit avoir $\overrightarrow{0}$ comme origine et ne pas nécessiter l'utilisation du modulo ce qui simplifie donc la formule précédente :

$$\left(M_{op}\right)_{\mathbf{M}} . \left(O_{op}\right)_{\mathbf{S}} . \left|P_{op} \quad F_{op} \quad 0\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad F_{res}}}$$

Ainsi nous obtenons la représentation d'une tâche complète. Cependant une tâche comporte plusieurs demi-tâches opérandes et résultats ce que nous n'avons pas décrit ici. Nous nous limiterons pour l'instant aux tâches avec seulement deux demi-tâches : une opérande et un résultat. Le cas général est abordé dans l'encadré page 55 ; mais nous verrons ultérieurement que son utilisation peut être contournée.

2.4.2.3 Équivalence du formalisme ODT et du modèle ARRAY-OL

Au vu de cette représentation on est en droit de se poser une question : les descriptions ARRAY-OL et ODT sont elles équivalentes ? La réponse est non et ceci à plusieurs niveaux et pour diverses raisons.

– Le niveau global n'a pas d'équivalent dans les ODT. Les dépendances exprimées dans les ODT ne montrent pas comment se déroule l'application. De plus, la taille des tableaux, présente dans le modulo, constitue la seule information permettant de savoir si le tableau résultat d'une tâche est le tâbleau opérande d'une autre. Bien évidemment, cette information n'est pas suffisante.

Représentation d'une tâche ARRAY-OL *à tableaux multiples*

Supposons que nous ayons plusieurs demi-tâches opérandes et résultats, la construction de la représentation ODT d'une telle tâche se fait en « concaténant verticalement » les réprésentations des différentes demi-tâches :

$$\begin{pmatrix} M_{op,1} \\ M_{op,2} \\ \vdots \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} O_{op,1} \\ O_{op,2} \\ \vdots \end{pmatrix}_{\mathbf{S}} . \left| \begin{array}{cccccc|c} P_{op,1} & F_{op,1} & 0 & 0 & 0 & 0 & \\ P_{op,2} & 0 & F_{op,2} & 0 & 0 & 0 & \\ \vdots & 0 & 0 & \ddots & 0 & 0 & 0 \end{array} \right| . \begin{pmatrix} Q \\ D_{op,1} \\ D_{op,2} \\ \vdots \\ D_{res,1} \\ D_{res,2} \\ \vdots \end{pmatrix}_{\mathbf{G}}$$

$$. \left| \begin{array}{ccccccc} P_{res,1} & 0 & 0 & 0 & F_{res,1} & 0 & 0 \\ P_{res,2} & 0 & 0 & 0 & 0 & F_{res,2} & 0 \\ \vdots & 0 & 0 & 0 & 0 & 0 & \ddots \end{array} \right| . \begin{pmatrix} O_{res,1} \\ O_{res,2} \\ \vdots \end{pmatrix}_{\mathbf{S}} . \begin{pmatrix} M_{res,1} \\ M_{res,2} \\ \vdots \end{pmatrix}_{\star}$$

Cette forme « complète » de la représentation d'une tâche ARRAY-OL n'est qu'une extension de la forme précédente et s'explique par le même raisonnement (*cf* section 2.4.2 page 52). Les pavages sont concaténés car ils sont globaux à la tâche, en revanche les ajustages sont spécifiques à chaque demi-tâche, on les considère donc séparement. Un *shift* et un éclatement sont présents car une tâche a nécessairement une demi-tâche résultat exacte, mais elles ne le sont pas toutes dans certains cas particuliers.
Quelles sont les differences entre la réprésentation ODT et la représentation ARRAY-OL :
– les demi-tâches ne sont plus différenciables les unes des autres, la présence de zéro ne permettant pas de faire la distinction entre les différentes parties des matrices ;
– la notion de demi-tâche maîtresse a été perdu ;
Ces informations devront donc être retrouvées pour permettre la transcription inverse.

– le niveau local n'est, lui, pas complètement transcrit. Il n'y a plus de distinction possible entre le pavage et l'ajustage une fois les valeurs numériques appliquées. Les informations concernant le traitement effectué par la tâche n'apparaissent plus ; le type des données traitées par la tâche est également absent. De plus, lorsqu'il y a plusieurs tableaux d'entrée ou de sortie, le passage à la représentation ODT pose un certain nombre de problèmes supplémentaires comme le montre l'encadré page 55. Ainsi, la représentation ODT d'une tâche ne permet pas de revenir à sa description ARRAY-OL sans qu'un certain nombre d'informations n'ait été conservé.

Il n'y a donc pas équivalence des deux représentations. Le formalisme ODT n'est qu'un outil de calcul permettant d'effectuer un certain nombre d'opérations et ne constitue en rien un modèle complet des applications ARRAY-OL.

2.4.2.4 Propriétés induites par ARRAY-OL

À l'aide des égalités mathématiques décrites dans la section 1.3.3.1 page 28, il est possible de faire quelques déductions simples mais fort utiles sur la représentation ODT d'une tâche. Il est ainsi possible de vérifier rapidement si une représentaion ODT est correcte ou non.

Soit une tâche T

$$(M_{op})_{\mathbf{M}}.(O_{op})_{\mathbf{S}}.\left|P_{op} \quad F_{op} \quad 0\right|.\begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}}.\overline{\underline{P_{res} \quad 0 \quad F_{res}}} \quad .$$

Rappelons que le modulo, le *shift* et le gabarit sont des vecteurs, alors que la projection et la segmentation sont des matrices. On montre facilement que :

- le modulo, le *shift* et la projection ont le même nombre de lignes qui est également le nombre de dimensions du ou des tableaux opérandes ;
- la segmentation a autant de lignes que le ou les tableaux résultats ont de dimensions ;
- la taille du gabarit est égale au nombre de colonnes de la projection et de la segmentation ;
- la partie de pavage de la projection (P_{op}) a autant de colonnes que celle de la segmentation (P_{res}). Mais ce nombre est également à la taille du gabarit de pavage Q ;
- le raisonnement précédent est applicable aux parties d'ajustage opérande et résultat dont le nombre de colonnes concorde avec celui des matrices de zéros qui leurs sont associées, et qui concorde également respectivement avec le gabarit d'ajustage opérande et résultat D_{op} et D_{res} ;
- a contrario, le *shift*, le modulo, et la projection peuvent ne pas avoir le même nombre de lignes que la segmentation.

2.4.3 Calculs sur les ODT

Avant d'approfondir et de démontrer comment il est possible d'optimiser une application ARRAY-OL, nous allons lister ici un certain nombre de résultats intermédiaires issus de calculs sur les ODT et qui serviront lors de la démonstration de la « fusion » qui est l'épine dorsale du processus d'optimisation. La plupart de ces résultats sont issus de la thèse de Julien Soula, mais j'ai dû refaire au début de mon doctorat tous les calculs permettant de les obtenir et ceci afin d'avoir une démonstration claire et complète de la fusion. Ces calculs ne sont pas tous présentés ici, mais ils sont décrits en détails dans **[**24**]**.

2.4.3.1 Opérations simples

Ces premières opérations sont facilement démontrables par le dessin ou le calcul et ne seront pas donc pas détaillées.

Composition d'ODT de même paramètre : $(\overrightarrow{x})_{\mathbf{M}}, (\overrightarrow{x})_{\star}, (\overrightarrow{x})_{\mathbf{G}}$

$\swarrow$.	**M**	$\star$	**G**
M	**M**	**G**	**G**
$\star$	$\star\star$	$\star$	$\star$
G	**M**	**G**	**G**

Autres compositions d'ODT :

$$\left|\mathcal{M}_1\right|.\left|\mathcal{M}_2\right| = \left|\mathcal{M}_1 \times \mathcal{M}_2\right|$$
$$\left|\mathcal{M}_1\right|.(\mathcal{S})_{\mathbf{S}} = (\mathcal{M}_1 \times \mathcal{S})_{\mathbf{S}}.\left|\mathcal{M}_1\right|$$

Création et suppression de dimensions : Afin de supprimer les dimensions d'ajustages ajoutées au gabarit, nous avons fait correspondre à chacune de ces dimensions un vecteur nul dans la projection. Or il est possible de généraliser cette utilisation en se servant de la projection comme d'un « suppresseur de dimension » et donc de la segmentation comme d'un « créateur de dimension ». Pour supprimer une dimension, il suffit de mettre à zéro le vecteur qui lui est associé dans la matrice de projection. De plus, pour laisser intact les autres dimensions il suffit d'utiliser des vecteurs de la taille du nouvel espace souhaité, ces derniers devant former la matrice identité si on supprime le ou les vecteurs à zéro (*cf* équation 2.1). Pour la segmentation, le principe est le même ; la construction de la matrice est identique. Mais les dimensions nouvellement créées contiennent toutes les valeurs possibles sur $\mathbb{K}$ (*cf* équation 2.2).

$$\begin{pmatrix} x \\ z \\ t \end{pmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \blacktriangleleft \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} \quad . \qquad (2.1)$$

$$\begin{pmatrix} x \\ \mathbb{K} \\ z \\ t \end{pmatrix} = \begin{array}{|cccc|} \hline 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline \end{array} \blacktriangleleft \begin{pmatrix} x \\ z \\ t \end{pmatrix} \quad . \qquad (2.2)$$

Dans un soucis de simplification, on réordonne les dimensions de telle façon que les vecteurs non nuls se retrouvent en tête de la matrice (ce qui n'entraîne aucune perte de généralité). On peut alors utiliser les notations : $\left|\mathbf{1}\quad\mathbf{0}\right|$ ou $\underline{\mathbf{1}\quad\mathbf{0}}$.

Quelques compositions :

$$\underline{\mathbf{1}\quad\mathbf{0}}.\left(\overrightarrow{g}\right)_{\mathbf{G}} = \begin{pmatrix} \overrightarrow{g} \\ \sim \end{pmatrix}_{\mathbf{G}}.\underline{\mathbf{1}\quad\mathbf{0}}$$

$$\underline{\mathbf{1}\quad\mathbf{0}}.\left|\mathcal{P}\right| = \begin{vmatrix} \mathcal{P} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} \end{vmatrix}.\underline{\mathbf{1}\quad\mathbf{0}}$$ [2]

$$\underline{\mathbf{1}\quad\mathbf{0}}.\left(\overrightarrow{m}\right)_{\mathbf{M}} = \begin{pmatrix} \overrightarrow{m} \\ \sim \end{pmatrix}_{\mathbf{M}}.\underline{\mathbf{1}\quad\mathbf{0}}$$

$$\underline{\mathbf{1}\quad\mathbf{0}}.\left(\overrightarrow{s}\right)_{\mathbf{S}} = \begin{pmatrix} \overrightarrow{s} \\ \overrightarrow{0} \end{pmatrix}_{\mathbf{S}}.\underline{\mathbf{1}\quad\mathbf{0}}$$

$$\underline{\mathbf{1}\quad\mathbf{0}}.\left(\overrightarrow{m}\right)_{\star} = \begin{pmatrix} \overrightarrow{m} \\ \sim \end{pmatrix}_{\star}.\underline{\mathbf{1}\quad\mathbf{0}}$$

$$\underline{\mathbf{1}\quad\mathbf{0}}.\underline{\mathcal{P}} = \underline{\mathcal{P}\quad\mathbf{0}}$$

Décalage de l'espace d'itération : Dans la section 2.4.1.1 page 49, nous avons défini le gabarit comme un couple de vecteurs $\overrightarrow{min}$ et $\overrightarrow{max}$. Cependant, à l'aide du *shift*, on peut se ramener facilement au cas le plus simple où $\overrightarrow{min} = \overrightarrow{0}$. Dans le cadre d'une demi-tâche ARRAY-OL, on obtient l'égalité suivante :

$$\left(M\right)_{\mathbf{M}}.\left(O\right)_{\mathbf{S}}.\left|\mathcal{M}\right|.\left(\overrightarrow{min},\overrightarrow{max}\right)_{\mathbf{G}} = \left(M\right)_{\mathbf{M}}.\left(O+\mathcal{M}\times\overrightarrow{min}\right)_{\mathbf{S}}.\left|\mathcal{M}\right|.\left(\overrightarrow{max}-\overrightarrow{min}\right)_{\mathbf{G}}$$

2.4.3.2 Opérations complexes

Première simplification : nous considérons ici une suite : projection, gabarit, segmentation. Il est possible de réduire le nombre de dimensions traitées dans trois cas distincts si $\mathbb{K} = \mathbb{Z}$:

[2] La taille de la matrice identité dans la segmentation n'est pas la même de part et d'autre de l'égalité. À gauche la matrice a pour taille le nombre de lignes de $\mathcal{P}$ et à droite le nombre de colonnes. De plus la matrice identité de la projection de droite a pour taille le nombre de colonnes de 0 dans la segmentation.

– La projection et la segmentation ont deux colonnes correspondantes à zéro. Il est donc possible de supprimer cette dimension dans les trois ODT puisqu'elle n'intervient jamais dans les calculs (créée par la segmentation, elle sera détruite par la projection après avoir subi un filtrage inutile par le gabarit).

$$\begin{vmatrix} 2 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 \\ 7 & 0 & 0 & 8 \end{vmatrix} . \begin{pmatrix} 7 \\ 15 \\ 0 \\ 1 \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\begin{matrix} 7 & 0 & 0 & 6 \\ 1 & 0 & 8 & 0 \end{matrix}}} = \begin{vmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 7 & 0 & 8 \end{vmatrix} . \begin{pmatrix} 7 \\ 0 \\ 1 \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\begin{matrix} 7 & 0 & 6 \\ 1 & 8 & 0 \end{matrix}}}$$

– Le gabarit a une valeur à zéro, il ne laisse donc rien passer sur cette dimension et elle n'intervient alors pas dans le calcul.

$$\begin{vmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 7 & 0 & 8 \end{vmatrix} . \begin{pmatrix} 7 \\ 0 \\ 1 \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\begin{matrix} 7 & 0 & 6 \\ 1 & 8 & 0 \end{matrix}}} = \begin{vmatrix} 2 & 0 \\ 0 & 0 \\ 7 & 8 \end{vmatrix} . \begin{pmatrix} 7 \\ 1 \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\begin{matrix} 7 & 6 \\ 1 & 0 \end{matrix}}}$$

– Le gabarit a une valeur à un, il ne laisse donc rien passer si ce n'est zéro par conséquent là encore cette dimension n'intervient pas dans le calcul.

$$\begin{vmatrix} 2 & 0 \\ 0 & 0 \\ 7 & 8 \end{vmatrix} . \begin{pmatrix} 7 \\ 1 \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\begin{matrix} 7 & 6 \\ 1 & 0 \end{matrix}}} = \begin{vmatrix} 2 \\ 0 \\ 7 \end{vmatrix} . (7)_{\mathbf{G}} . \overline{\underline{7 \\ 1}}$$

Deuxième simplification : Cette nouvelle simplification s'applique à une suite projection - gabarit - segmentation. Si on construit une seule matrice à partir du pavage de la projection et de la segmentation concernées $\begin{pmatrix} P_{op} \\ P_{res} \end{pmatrix}$, alors il est possible de supprimer les vecteurs colinéaires de cette matrice sous certaines conditions. Il suffit ensuite de séparer la matrice afin d'obtenir une expression simplifiée. Il est possible de recommencer la simplification avec l'ajustage opérande puis avec l'ajustage résultat. Cette simplication n'est que très peu décrite car elle n'est pas indispensable à la compréhension des optimisations.

Calcul d'une « bouding-box » : Soit $\overrightarrow{min}$ et $\overrightarrow{max}$ les deux extremums permettant d'englober les points générés par une suite *shift* - projection - gabarit $((\overrightarrow{s})_{\mathbf{S}} . |\mathcal{M}| . (\overrightarrow{g})_{\mathbf{G}})$ avec $\mathbb{K} = \mathbb{Q}$.

Tous les points générés par notre expression sont donc compris dans la boîte englobante définie par $\overrightarrow{min}$ et $\overrightarrow{max}$. Or ces points ne sont pas nécessairement entiers car $\mathcal{M}$ est définie sur $\mathbb{Q}$ et nos extremums peuvent eux aussi avoir une valeur fractionnaire. Nous allons construire une nouvelle expression générant tous les points entiers compris dans la boîte englobante définie par les valeurs arrondies de $\overrightarrow{min}$ et $\overrightarrow{max}$. Cette expression ($\overrightarrow{min}$ et $\overrightarrow{max}$ ayant été préalablement arrondis) s'obtient à l'aide d'un décalage et d'un parcours de l'espace d'itération : $\left(\overrightarrow{min}\right)_{\mathbf{S}} . |\mathbf{1}| . \left(\overrightarrow{max} + \overrightarrow{1} - \overrightarrow{min}\right)_{\mathbf{G}}$.

L'inconvénient principal de cette boîte est qu'elle augmente considérablement le nombre de points générés (*cf* encadré page 59).

Points supplémentaires générés par une boite englobante

Soit :
$(\overrightarrow{s})_{\mathbf{S}} . |\mathcal{M}| . (\overrightarrow{g})_{\mathbf{G}} = (0)_{\mathbf{S}} . |5| . (3)_{\mathbf{G}}$
Les points générés sont (0), (5) et (10).
Donc si on calcule la boite englobante, on obtient : $(0)_{\mathbf{S}} . |1| . (11)_{\mathbf{G}}$
Les points générés sont (0), (1), (2), (3), (4), ... , (10).
NB : Plus les vecteurs de $\mathcal{M}$ seront petits, moins il y aura de nouveaux points générés en trop

Segmentation du gabarit : Soient $\overrightarrow{g}$ et $\overrightarrow{\lambda}$ deux vecteurs tels que $\forall\, i, \lambda_i$ divise g_i, alors

$$\begin{pmatrix} g_1 \\ \vdots \\ g_n \end{pmatrix}_{\mathbf{G}} = \begin{vmatrix} \lambda_1 & & & 1 & & \\ & \ddots & & & \ddots & \\ & & \lambda_n & & & 1 \end{vmatrix} . \begin{pmatrix} g_1/\lambda_1 \\ \vdots \\ g_n/\lambda_n \\ \lambda_1 \\ \vdots \\ \lambda_n \end{pmatrix}_{\mathbf{G}} \cdot \overline{\begin{matrix} \lambda_1 & & & 1 & & \\ & \ddots & & & \ddots & \\ & & \lambda_n & & & 1 \end{matrix}}$$

Ce résultat est facilement démontrable par le calcul. Cette opération s'appelle segmentation car elle réalise l'opération inverse de la linéarisation en dédoublant l'expression d'une dimension comme sur l'exemple de la figure 2.1.

$$(10)_{\mathbf{G}} = \begin{vmatrix} 2 & 1 \end{vmatrix} \begin{pmatrix} 5 \\ 2 \end{pmatrix}_{\mathbf{G}} \overline{\begin{matrix} 2 & 1 \end{matrix}}$$

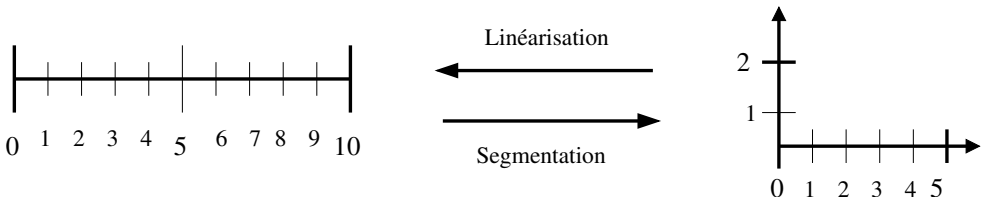


FIG. 2.1: Segmentation du gabarit

Cette opération a un double rôle qui trouve tout son avantage dans le cadre d'ARRAY-OL. Soit une demi-tâche dont on ne conserve que la projection et le gabarit : $\begin{vmatrix} P & F \end{vmatrix} . \left(\begin{smallmatrix} Q \\ D \end{smallmatrix}\right)_G$. Si on effectue une segmentation du gabarit uniquement sur la partie de pavage de ce dernier, on réduit le nombre de motifs à calculer. En effet, le nombre d'itérations est divisé par λ_i pour chaque vecteur, et on passe le reste de l'itération dans l'ajustage ce qui a pour effet d'augmenter la taille du motif[3]. On obtient alors :

$$\begin{vmatrix} P & F \end{vmatrix} . \begin{pmatrix} Q \\ D \end{pmatrix}_{\mathbf{G}} = \begin{vmatrix} P & F \end{vmatrix} . \begin{vmatrix} \Lambda & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} . \begin{pmatrix} Q \\ \lambda \\ D \end{pmatrix}_{\mathbf{G}} \cdot \overline{\begin{matrix} \Lambda & 1 & 0 \\ 0 & 0 & 1 \end{matrix}} = \begin{vmatrix} P \times \Lambda & P & F \end{vmatrix} . \begin{pmatrix} Q \\ \lambda \\ D \end{pmatrix}_{\mathbf{G}} \cdot \overline{\begin{matrix} \Lambda & 1 & 0 \\ 0 & 0 & 1 \end{matrix}}$$

[3]Le lecteur opposera à cette solution qu'elle nécessite une modification de la tâche élémentaire elle-même, mais nous verrons dans le cadre de l'utilisation que ce n'est pas important.

Or si les valeurs de la projection ne sont pas entières[4], il est possible de supprimer les valeurs fractionnaires de la partie de pavage en choisissant astucieusement les valeurs de λ. Pour cela on choisit tous les λ_i tels que $\lambda_i.P_{*,i}$ soient entiers et si λ_i ne divise pas q_i alors on prend $\lambda_i = q_i$ ce qui revient à passer toute la dimension dans l'ajustage. En effet, si $\lambda_i = q_i$ alors $\lambda_i/q_i = 1$ et donc on peut appliquer la première simplification.

La segmentation a donc une double utilité, mais chacune intervient à un niveau de lecture différent du formalisme : la première intervient au niveau de la représentation ARRAY-OL en réduisant le nombre de calcul, la deuxième au simple niveau mathématique en supprimant des valeurs fractionnaires dans la projection.

Une des grandes difficultés d'utilisation des ODT réside dans la nécessité d'une lecture à plusieurs niveaux.

2.4.3.3 Opérations dans le cadre d'ARRAY-OL

ODT exact : un ODT exact représente une demi-tâche résultat exacte. La projection et le gabarit d'un tel ODT ont des propriétés qui découlent de celles des demi-tâches exactes. Si on se place dans le cadre d'une tâche complète, le gabarit et la projection ont été modifiés pour la gestion de la concordance des espaces au niveau du gabarit, il est alors possible de démontrer que :

$$\left(\begin{vmatrix} P_{res} & 0 & F_{res} \end{vmatrix} . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}}\right)^{-1} = \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} P_{res} & 0 & F_{res} \end{matrix}} = \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{M}} . \lfloor . \overline{\mathcal{M}''} . \begin{vmatrix} {P_{res}}' \\ 0 \\ {F_{res}}' \end{vmatrix} .$$

Avec $\mathcal{M}''$ une matrice diagonale portant un 1 si la ligne correspondante de $\mathcal{M}'$ (ici $\begin{vmatrix} {P_{res}}' \\ 0 \\ {F_{res}}' \end{vmatrix}$) est non nulle et $\mathcal{M}'$ obtenu par transposition et inversion des termes non nuls de $\mathcal{M}$ (ici $\begin{vmatrix} P_{res} & 0 & F_{res} \end{vmatrix}$)

Le point le plus important de cette égalité est qu'elle introduit des valeurs fractionnaires dues aux inversions dans la projection. Or ces valeurs ne sont pas correctes en ARRAY-OL, il faudra donc les supprimer pour revenir à une expression correcte.

Suppression du modulo : il est parfois utile de pouvoir supprimer des modulos qui s'avèrent génants. Dans le cas de l'ODT : $(\overrightarrow{m_2})_{\mathbf{M}}.\left|\mathcal{M}\right|.(\overrightarrow{m_1})_{\mathbf{M}'}$ il est possible de supprimer le modulo sous certaines conditions dans un cadre général et dans tous les cas dans le cadre d'une ODT exacte. Il est ainsi possible de démontrer que :

– pour le cas général :

$$\text{si} \forall m_i \in \overrightarrow{m_1}, (\overrightarrow{m_2})_{\mathbf{M}}.\left|\mathcal{M}\right| \blacktriangleleft \begin{pmatrix} 0 \\ \vdots \\ m_i \\ \vdots \\ 0 \end{pmatrix} = \overrightarrow{0} \text{ alors } (\overrightarrow{m_2})_{\mathbf{M}}.\left|\mathcal{M}\right|.(\overrightarrow{m_1})_{\mathbf{M}} = (\overrightarrow{m_2})_{\mathbf{M}}.\left|\mathcal{M}\right|$$

[4]Ce qui est, certes, théoriquement impossible dans le cadre d'ARRAY-OL.

– pour le cas particulier :

$$\left(\overrightarrow{g}\right)_{\mathbf{M}} . \lfloor . \left|\mathcal{M}'\right| . \left(\overrightarrow{d}\right)_{\mathbf{M}} = \left(\overrightarrow{g}\right)_{\mathbf{M}} . \lfloor . \left|\mathcal{M}'\right|$$

Court circuit : notre but est ici d'exprimer les coordonnées des éléments d'un motif d'une sous-tâche selon les coordonnées des éléments du tableau de la tâche supérieure. On note de la façon suivante les représentations ODT des tâches supérieures et inférieures :

$$\left(O_{up}\right)_{\mathbf{S}} . \left|P_{up} \quad F_{up}\right| . \begin{pmatrix} Q_{up} \\ D_{up} \end{pmatrix}_{\mathbf{G}}$$

$$\left(O_{sub}\right)_{\mathbf{S}} . \left|P_{sub} \quad F_{sub}\right| . \begin{pmatrix} Q_{sub} \\ D_{sub} \end{pmatrix}_{\mathbf{G}}$$

On peut alors écrire que les coordonnées, pour des itérations de pavage et d'ajustage fixées, sont de la forme :

$$\left(O_{up}\right)_{\mathbf{S}} . \left|P_{up} \quad F_{up}\right| . \left(\begin{matrix} X_{q,up} \\ O_{sub} + \left|P_{sub} \quad F_{sub}\right| . \begin{pmatrix} X_q \\ X_d \end{pmatrix}_{\mathbf{G}} \end{matrix} \right)_{\mathbf{G}}$$

On obtient donc des coordonnées égales à :

$$O_{up} + P_{up}X_{q,up} + F_{up}O_{sub} + \left|F_{up}P_{sub} \quad F_{up}F_{sub}\right| \begin{pmatrix} X_q \\ X_d \end{pmatrix}_{\mathbf{G}}$$

Il est alors possible de modifier la forme de nos deux tâches, en passant directement à la sous-tâche, un tableau qui corresponde aux motifs quelle consomme.

$$\left(O_{up} + F_{up}O_{sub}\right)_{\mathbf{S}} . \left|P_{up} \quad F_{up}P_{sub} \quad F_{up}F_{sub}\right| . \begin{pmatrix} Q_{up} \\ Q_{sub} \\ D_{sub} \end{pmatrix}_{\mathbf{G}}$$

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} . \begin{pmatrix} Q_{sub} \\ D_{sub} \end{pmatrix}_{\mathbf{G}}$$

2.4.4 Conclusion

Le formalisme ODT est bien adapté à l'expression des dépendances au sein d'une tâche ARRAY-OL et est d'une utilisation relativement simple. Cependant il n'offre pas une représentation explicite de la tâche et les modifications sur les ODT contraignent le lecteur à calculer une représentation visuelle des modifications. Il faut donc adopter une lecture à plusieurs niveaux pour appréhender parfaitement le formalisme et ses manipulations. De plus, les ODT ne permettent pas d'exprimer les dépendances entre les itérations de pavages, il n'est donc pas possible avec ce formalisme d'exprimer les états et les délais tels qu'ils sont décrits dans GMDSDF.

2.5 Conclusion

Dans ce chapitre, nous avons étudié différentes méthodes pouvant mener à l'optimisation d'une application ARRAY-OL. Nous avons justifié le choix des ODT en analysant les possibilités offertes par les techniques de transformations de boucles. Nous avons alors montré que l'expression des dépendances se trouve au centre des problèmes d'optimisation et que les ODT sont parfaitement adaptés pour les exprimer. Enfin, nous avons présenté un certain nombre de résultats issus de calculs sur ces ODT et qui nous serviront dans le chapitre suivant.

Chapitre 3

Transformation des applications ARRAY-OL

Labor omnia vicit inprobus.

Virgile, *Les Géorgiques* L1 (145-146).

3.1 Introduction

Nous venons d'étudier dans le chapitre précédent le formalisme ODT dans le but de transformer les applications ARRAY-OL. Les premiers travaux sur ces transformations ont été réalisés par Julien Soula au cours de sa thèse [59, 58]. J'ai par la suite repris ses travaux et je les ai complétés afin de fournir une « boîte à outils » capable d'adapter la forme de l'application à celle de l'architecture en optimisant différents critères par l'enchaînement « intelligent » des transformations. Nous étudierons successivement dans ce chapitre les différentes transformations qui ont été mises au point.

3.2 Fusion

Notre première tâche est de réaliser l'optimisation, à l'aide du formalisme ODT, que nous avons décrit à la section 2.3.2.3 page 47. Nous appelons cette optimisation « fusion » bien qu'elle n'ait qu'un rapport indirect avec la fusion de boucles. La fusion consiste à réduire deux tâches en une seule, la nouvelle tâche faisant appel à deux sous-tâches. La figure 3.1 montre les deux tâches avant la fusion et la figure 3.2 montre le résultat de la fusion.

3.2.1 Principe de la fusion

La nouvelle tâche $T3$ réalise en une seule fois ce que faisaient les tâches $T1$ et $T2$. Elle consomme en entrée le tableau $A1$ et produit en sortie le tableau $A3$. Il y a donc conservation des entrées sorties, la fusion permet de remplacer les deux tâches sans modifier le reste de l'application.

$T3$ est une tâche hiérarchique qui est composée de deux sous-tâches : $T1$ et $T2$ qui réalisent le même traitement qu'auparavant mais dont les tableaux d'entrée et de sortie dif-

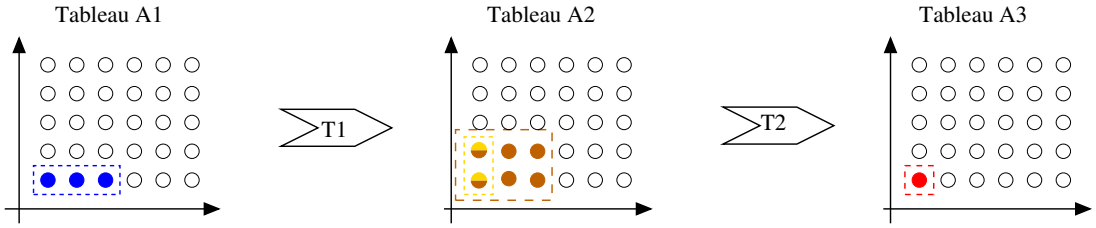


FIG. 3.1: Avant la fusion

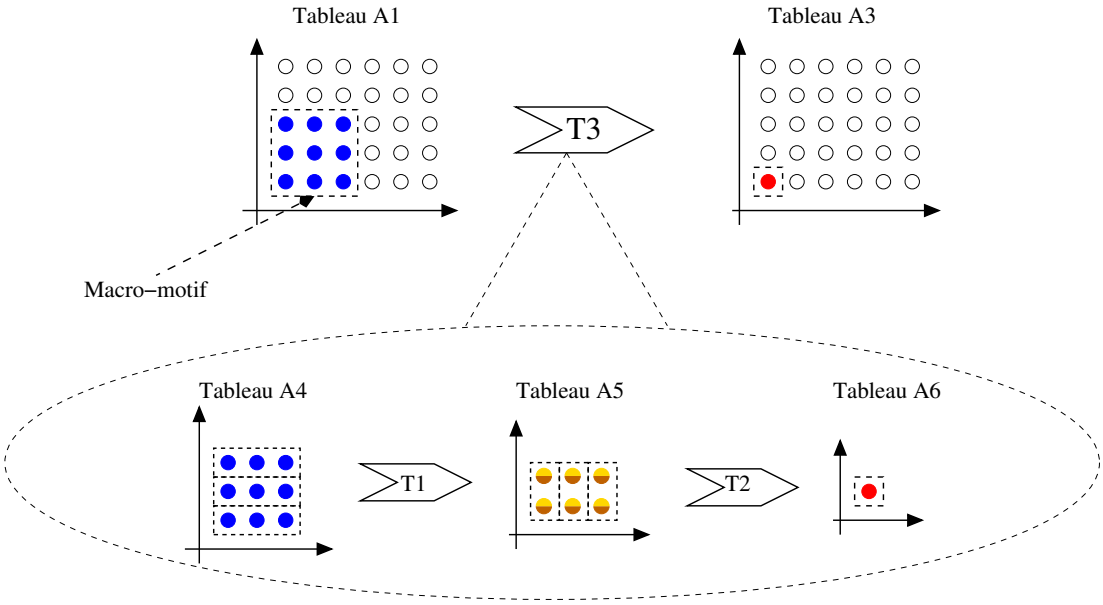


FIG. 3.2: Après la fusion

fèrent. En effet, on ne peut se contenter d'une seule tâche car il faut pouvoir appliquer les traitements réalisés par $T1$, et par $T2$.

Le but est de produire à chaque itération au moins un motif du tableau $A3$. Or pour obtenir un motif de $A3$, un ou plusieurs motifs de $A1$ sont nécessaires. C'est pour cela que le tableau $A4$ est constitué par l'agglomération de plusieurs motifs de $A1$, de manière à ce qu'ils produisent après leur passage dans $T1$ suffisamment d'éléments pour remplir au moins un motif opérande complet de $T2$. Cette agglomération est appelée macro-motif.

La réalisation d'une fusion passe par la quête des macro-motifs opérandes mais aussi résultats car nous verrons que dans certains cas nous produirons à chaque itération plusieurs motifs. Pour atteindre notre objectif, nous allons utiliser le formalisme ODT et sa capacité à exprimer les dépendances.

On représente $T1$ et $T2$ par leur forme ODT :

$$T_1 \mapsto \left(M_1\right)_{\mathbf{M}} . \left(S_1\right)_{\mathbf{S}} . \left|P_{op,1} \quad F_{op,1} \quad 0\right| . \begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,1} \quad 0 \quad F_{res,1}}} \ ,$$

$$T_2 \mapsto \left(M_2\right)_{\mathbf{M}} . \left(S_2\right)_{\mathbf{S}} . \left|P_{op,2} \quad F_{op,2} \quad 0\right| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,2} \quad 0 \quad F_{res,2}}} \ .$$

La représentation ODT d'une tâche se décompose en deux parties : le passage de l'ensemble des coordonnées du tableau opérande à l'espace QD et le passage de l'espace QD au tableau résultat. Si on concatène ces deux tâches on voit que le passage de l'espace QD de $T1$ à celui de $T2$ (c'est-à-dire de Q_1D_1 à Q_2D_2) passe par $A2$.

$$\left(M_1\right)_{\mathbf{M}} . \left(S_1\right)_{\mathbf{S}} . \left|P_{op,1} \quad F_{op,1} \quad 0\right| . \underbrace{\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,1} \quad 0 \quad F_{res,1}}} . \left(M_2\right)_{\mathbf{M}} . \left(S_2\right)_{\mathbf{S}} . \left|P_{op,2} \quad F_{op,2} \quad 0\right| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}}}_{\text{relation } Q_1D_1 \rightarrow Q_2D_2 \text{ passant par } A_2} . \overline{\underline{P_{res,2} \quad 0 \quad F_{res,2}}} \qquad (3.1)$$

Notre but est de transformer la partie de l'expression précédente entre accolade afin d'obtenir une nouvelle expression qui ne devra plus passer par $A2$ mais qui devra formaliser les dépendances qui existent entre Q_1D_1 et Q_2D_2 (pour connaître ce qu'il est nécessaire de consommer de $A1$ pour produire au moins un motif de $A3$). Or si par le biais de la transformation nous obtenons une nouvelle tâche ARRAY-OL, alors notre expression représente naturellement les dépendances entre les espaces de départ et d'arrivée que sont Q_1D_1 et Q_2D_2.

Afin de mieux comprendre ce raisonnement, il faut étudier la forme de cette expression et les corrélations entre les différents espaces mis en jeux.

Cette tâche part de Q_1D_1 et va dans Q_2D_2 en passant par un nouvel espace QD appelé « super QD ».

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \left(S\right)_{\mathbf{S}} . \left|P_{op} \quad F_{op} \quad 0\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad F_{res}}} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\star}$$

Regardons comment interpréter les différents éléments de cette tâche. À chaque itération de pavage (itération sur Q), la nouvelle tâche consomme un motif opérande sur Q_1D_1 et produit un motif résultat sur Q_2D_2. Mais qu'est ce qu'un motif sur un espace d'itération ?

Dans un espace d'itération, il y a les dimensions d'ajustage et les dimensions de pavage, celles d'ajustage désignent des points et celles de pavages des motifs. Si nous prenons un motif dans cet espace, c'est-à-dire un ensemble de points sur les différentes dimensions, nous allons en fait désigner des points dans des motifs des espaces de départ et d'arrivée. Plus précisement, on ne désigne des points que dans un seul motif si les coordonnées des points de l'espace d'itération ne varient pas sur les dimensions de pavage. Et si l'ensemble de

ces points englobent toutes les dimensions d'ajustages, alors nous allons désigner un motif complet (*cf* figure 3.3 [1]).

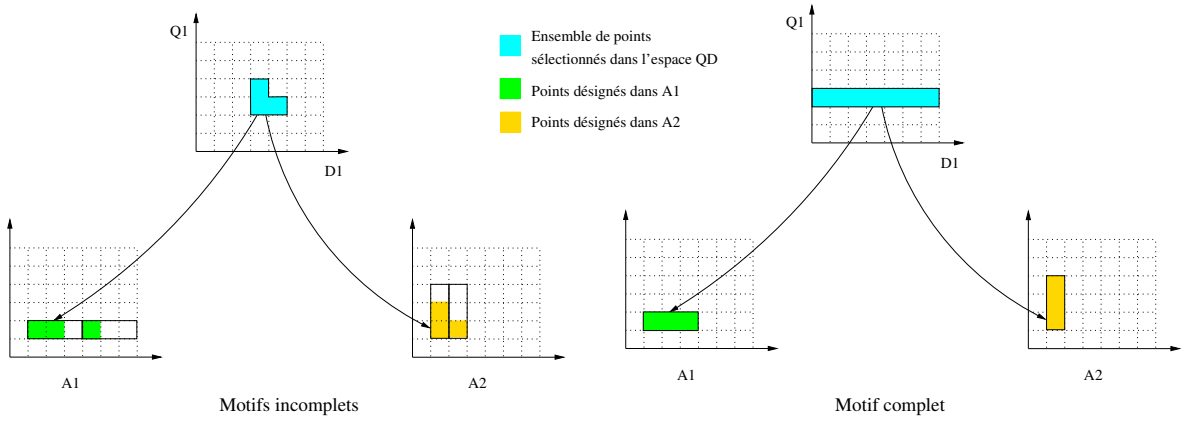


FIG. 3.3: Désignation des motifs de $A1$ et $A2$ via l'espace Q_1D_1

Dans notre cas, il faut qu'à chaque itération sur Q, nous désignons suffisamment de points dans Q_1D_1 pour pouvoir construire le macro-motif dans $A1$. Une itération sur Q est une itération de pavage du macro-motif, on parle de macro-pavage. Les points désignés dans Q_1D_1, doivent englober toutes les dimensions d'ajustages puisqu'ils désignent eux-même des motifs dans $A1$ (*cf* figure 3.4 page 68[1]).

Les itérations d'ajustage sur D servent à construire le macro-motif, il n'existe pas de contraintes prédéfinies sur cet ajustage. Toutefois il est possible de le subdiviser en deux parties : l'ajustage spatial et le macro-ajustage. En effet comme le macro-motif est constitué de motifs originaux de $A1$, on peut envisager sa construction en deux étapes :

- construction d'un motif original par itération sur l'ajustage spatial ;
- passage d'un motif original à l'autre par itération sur le macro-ajustage.

L'ajustage spatial est « chargé » de la construction du motif, donc une itération complète des valeurs de l'ajustage spatial doit désigner des points qui englobent exactement toutes les dimensions d'ajustage dans Q_1D_1. Le macro-ajustage, quant à lui, prend la place du pavage original. Les itérations sur le macro-ajustage servent à désigner des points avec des coordonnés de pavage différentes dans Q_1D_1.

La totalité des itérations d'ajustage sur D permettent de construire un motif dans Q_1D_1 et donc le macro-motif dans $A1$. Et pour mémoire les itérations de pavage sur Q permettent, elles, d'itérer le motif de Q_1D_1 donc le macro-motif.

Ce raisonnement s'applique symétriquement à la partie résultat de notre nouvelle tâche et est illustré par les figures 3.5 page 68[1] et 3.4 page 68[1]. À partir de ces réflexions, on peut affiner le squelette de notre expression une fois transformée ($\mathcal{M}$ en indice indique un macro-ajustage, et l'absence d'indice désigne un ajustage spatial et non plus l'ajustage dans son ensemble).

[1]Les figures 3.3, 3.4 page 68 et 3.5 page 68 sont simplifiées : les parties opérandes et résultats de l'espace d'itération D ne sont pas distinguées. La taille de ce dernier n'est donc pas représentative. Il faut simplement distinguer les cas où on le prend dans sa globalité pour avoir des motifs complets des cas où il ne sert que partiellement.

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . (S)_{\mathbf{S}} . \left| P_{op} \quad F_{\mathcal{M},op} \quad F_{op,1} \quad 0 \quad 0 \right| . \begin{pmatrix} Q \\ D_{\mathcal{M},op} \\ D_{op,1} \\ D_{\mathcal{M},res} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad 0 \quad F_{\mathcal{M},res} \quad F_{res,2}}} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\star}$$

Résumons, si notre nouvelle tâche est une tâche ARRAY-OL nous savons qu'elle consommera suffisamment de motifs de $A1$ pour pouvoir en produire dans $A3$, mais afin qu'elle désigne de façon correcte ces motifs, il faut qu'à chaque itération d'ajustage (sur D) et plus précisement d'ajustage spatiale elle prenne les dimensions d'ajustages (D_1 et D_2) de ses espaces de départs $Q_1 D_1$ et d'arrivées $Q_2 D_2$ complètement.

3.2.2 Calcul de la fusion

Nous allons aborder maintenant le calcul de la fusion. Ce dernier fait office de démonstration et prouve que le résultat est correct, mais il indique aussi quelles sont les étapes à suivre pour obtenir une tâche fusionnée. Ainsi l'implémentation de la fusion telle qu'elle est décrite dans la section 3.2.4 page 73 reprend point par point les différentes étapes du calcul. Une compréhension détaillée de la fusion est donc primordiale pour quiconque veut travailler sur l'implémentation. Le calcul de la fusion est dû à Julien Soula, il n'est pas détaillée directement dans cette section et a été mis dans l'annexe B page 125. La compléxité et la longueur des opérations dépassent largement leurs intérets dans le cadre de cette thèse. Toutefois, nous présentons ici les principales étapes du raisonnement qui amène aux résultats et ceci afin de mieux comprendre ce qui suivra.

Le calcul de la fusion s'effectue en plusieurs étapes. La première étape permet d'obtenir directement une tâche ARRAY-OL. Mais cette dernière comporte des parties fractionnaires qu'on supprime lors de la deuxième étape. Puis il faut construire, à partir de la tâche ARRAY-OL obtenue, une nouvelle tâche hiérarchique.

3.2.2.1 Obtention d'une tâche ARRAY-OL

Le point de départ est donc la partie centrale de l'expression ODT des tâches une et deux mises bout à bout :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{G}} \overline{\underline{P_{res,1} \quad 0 \quad F_{res,1}}} \, (M_2)_{\mathbf{M}} (S_2)_{\mathbf{S}} \left| P_{op,2} \quad F_{op,2} \quad 0 \right| \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}}$$

Pour transformer cette expression en la représentation d'une tâche ARRAY-OL, on commence par utiliser les propriétés sur l'inversion des ODT exacts (*cf* section 2.4.3.3 page 60). Puis en combinant les matrices à l'aide des autres propriétés des ODT, on obtient aisément l'expression d'une tâche ARRAY-OL. Mais le *shift* et la projection de notre expression contiennent des valeurs fractionnaires dûes à l'utilisation de l'inversion des ODT exacts (*cf* 2.4.3.3 page 60). Avant de supprimer ces parties fractionnaires, on modifie notre expression pour qu'elle génère bien tous les points sur D et plus particulièrement dans l'ajustage spatial.

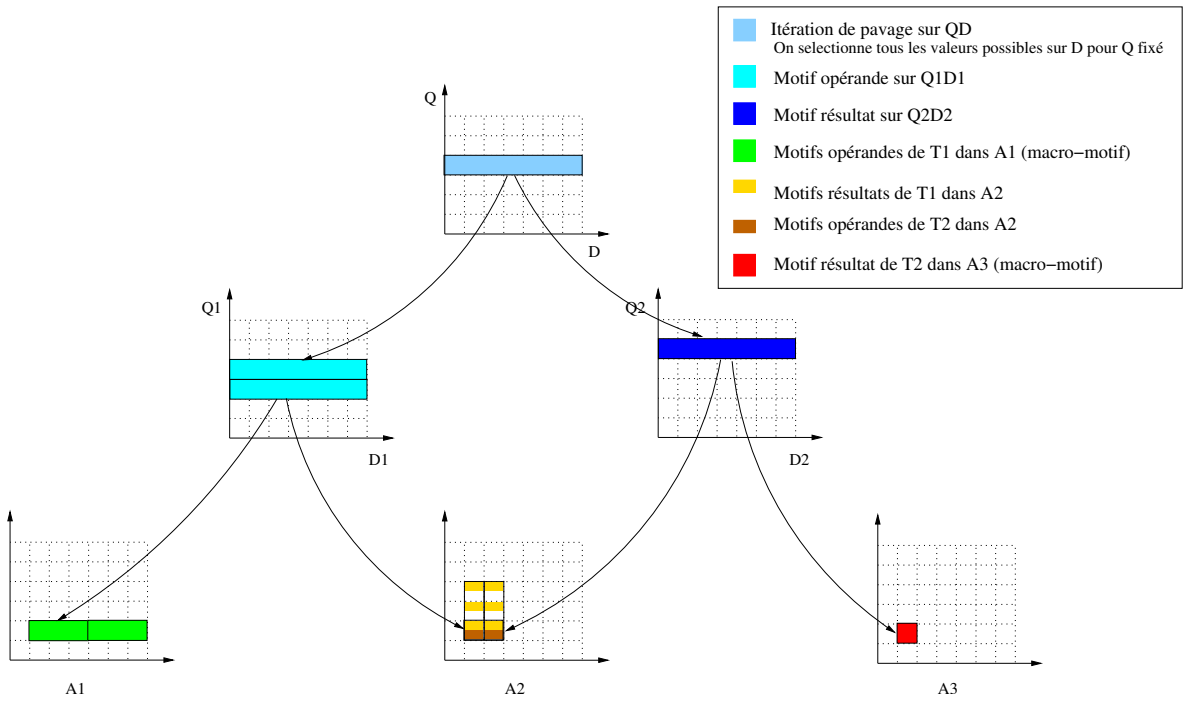


FIG. 3.4: Itération sur Q

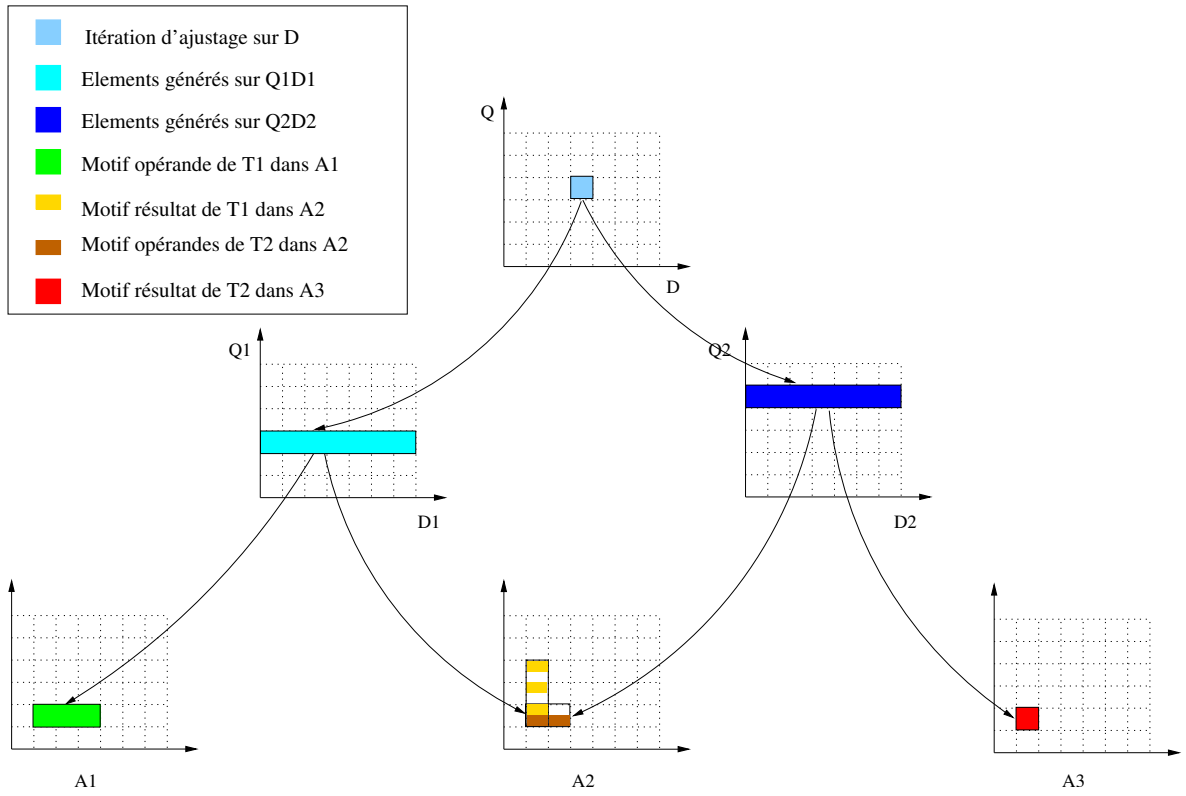


FIG. 3.5: Itération sur le macro-ajustage

3.2.2.2 Suppression des valeurs fractionnaires

Les valeurs fractionnaires se trouvant aussi bien dans le pavage que dans l'ajustage de la projection, nous allons commencer par utiliser la segmentation du gabarit pour éliminer les parties fractionnaires du pavage. Une fois le pavage devenu entier, il ne reste plus que l'ajustage. La méthode la plus simple pour supprimer les valeurs fractionnaires est de construire une boîte englobante autour des points qu'on souhaite manipuler. Mais cette méthode a comme inconvénient de générer beaucoup plus de points que nécessaire. (*cf* encadré page 59). Comme le montre cet encadré, plus la taille des vecteurs d'ajustage est grande plus le nombre de points générés en trop est important. Générer trop de points à partir de l'ajustage dans notre expression revient en fait à consommer trop de motifs opérande dans $A1$ pour produire un motif résultat dans $A3$. On décide donc de réduire ce nombre en séparant les parties entières et fractionnaires des vecteurs. On utilise alors cette inclusion :

$$\lfloor . (S)_{\mathbf{S}} . |F| . (D)_{\mathbf{G}} \subset (S)_{\mathbf{S}} . |Ent(F) \quad Frac(F)| . \begin{pmatrix} D \\ D \end{pmatrix}_{\mathbf{G}}$$

Mais nous devons faire face à un nouvel inconvénient puisqu'on élève le nombre d'itérations au carré, ce qui n'a pas d'impacts sur notre tâche en elle-même, mais cela risque de ralentir considérablement la construction des motifs par la suite. Nous utilisons donc un nouvel échappatoire en utilisant la segmentation de l'ajustage pour limiter la taille des bornes. En effet lorsqu'on effectue une segmentation du gabarit, que ce soit du gabarit de pavage ou du gabarit d'ajustage, la valeur de λ est choisie comme étant un diviseur des bornes. Or λ devient par la suite notre nouvelle borne pour les parties fractionnaires, donc le nombre d'itération des parties fractionnaires a été réduit.

Ainsi la suppression des valeurs fractionnaires s'effectue en quatre étapes :

1. segmentation du gabarit de pavage pour n'avoir à travailler que sur l'ajustage ;
2. segmentation du gabarit d'ajustage pour réduire la taille des bornes ;
3. séparation des parties entières et fractionnaires des vecteurs d'ajustage pour limiter leur taille ;
4. calcul d'une boîte englobante.

On obtient ainsi l'expression d'une tâche ARRAY-OL permettant de passer directement de $Q1D1$ à $Q2D2$ sans passer par $A2$. Cette tâche est donc appelée « super QD » et elle établit les dépendances entre $A1$ et $A3$.

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} \begin{pmatrix} S \\ 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \left| \begin{array}{ccccccc} P_{op} & F_{\mathcal{M},op} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \right| . \begin{pmatrix} Q \\ D_{\mathcal{M},op} \\ D_{op,1} \\ D_{res,1} \\ D_{\mathcal{M},res} \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \left| \begin{array}{ccccccc} P_{res} & 0 & 0 & 0 & F_{\mathcal{M},res} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right| .$$

3.2.2.3 Obtention de la hiérarchie

À partir de la « super QD », il est possible de construire la tâche hiérarchique telle qu'elle est décrite section 3.2.1 page 63.

L'opération est relativement facile pour la tâche supérieure puisqu'il suffit de remplacer la partie centrale de l'équation 3.1 page 65 par la « super QD » et d'utiliser la composition des ODT. Ce calcul soulève néanmoins un certain nombre de difficultés que nous ne détaillerons pas.

Nous avons donc calculé la valeur de la tâche supérieure, il faut donc que nous calculions les deux sous-tâches. Ces deux tâches sont équivalentes à $T1$ et $T2$ puisqu'elles doivent effectuer les mêmes traitements, et par conséquent, elles consomment et produisent les mêmes motifs. Or la première tâche de la hiérarchie doit consommer le macro-motif opérande de la tâche supérieure. De plus, ce macro-motif est l'agglomeration de plusieurs motifs de la tâche d'origine $T1$, il suffit donc d'utiliser une matrice identité pour paver et ajuster ce macro-motif dans la sous-tâche. Le raisonnement est le même pour montrer que la segmentation de la seconde tâche de la hiérarchie est également la matrice identité. Pour obtenir les dernières informations concernant les deux sous-tâches, on part d'un point se trouvant dans l'espace d'ajustage opérande ou résultat de notre tâche « super QD ». Ensuite, on parcourt « le chemin » reliant ce point au point correspondant de $A2$. Pour un point se trouvant dans l'ajustage opérande, on passe par la projection et le *shift* de la « super QD » pour arriver dans $Q_1 D_1$. Puis on passe par la segmentation de $T1$ pour arriver finalement dans $A2$. On compose donc ces ODT pour obtenir la première sous-tâche, et le raisonnement est le même pour la deuxième.

Les résultats obtenus sont de la forme :

– pour la tâche supérieure :

$$\left(M_1\right)_{\mathbf{M}} . \left(S_{up}\right)_{\mathbf{S}} . \left|P_{op,up} \quad F_{op,up} \quad F_{op,1} \quad 0 \quad 0\right| . \begin{pmatrix} Q \\ D_{\mathcal{M},op} \\ D_{op,1} \\ D_{\mathcal{M},res} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,up} \quad 0 \quad 0 \quad F_{res,up} \quad F_{res,2}}}$$

– pour la première sous-tâche :

$$\begin{pmatrix} D_{\mathcal{M},op} \\ D_{op,1} \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} . \begin{pmatrix} D_{\mathcal{M},op} \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,sub1} \quad 0 \quad F_{res,1}}} . \left(S_{res,sub1}\right)_{\mathbf{S}} . \left(M_2\right)_{\star}$$

– pour la deuxième sous-tâche :

$$\left(M_2\right)_{\mathbf{M}} . \left(S_{op,sub2}\right)_{\mathbf{S}} . \left|P_{res,sub2} \quad F_{op,2} \quad 0\right| . \begin{pmatrix} D_{\mathcal{M},res} \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}}$$

Nous avons finalement réussi à fusionner nos deux tâches originales en une même tâche hiérarchique comportant deux sous-tâches. Nous pouvons d'ores et déja remarquer que le résultat produit comporte des imperfections. En effet, la sous-tâche gauche, à cause de la présence d'un *shift* résultat, n'est pas exacte. De plus, rien ne prouve pour l'instant que la tâche supérieure soit elle même exacte (les matrices de pavages et d'ajustages, contenues dans sa segmentation peuvent très bien ne pas répondre aux critères de l'exactitude). En outre nous n'avons que la forme ODT de ces tâches et nous ne savons fusionner que des tâches simples qui ne comportent pas plusieurs tableaux d'entrées et de sorties.

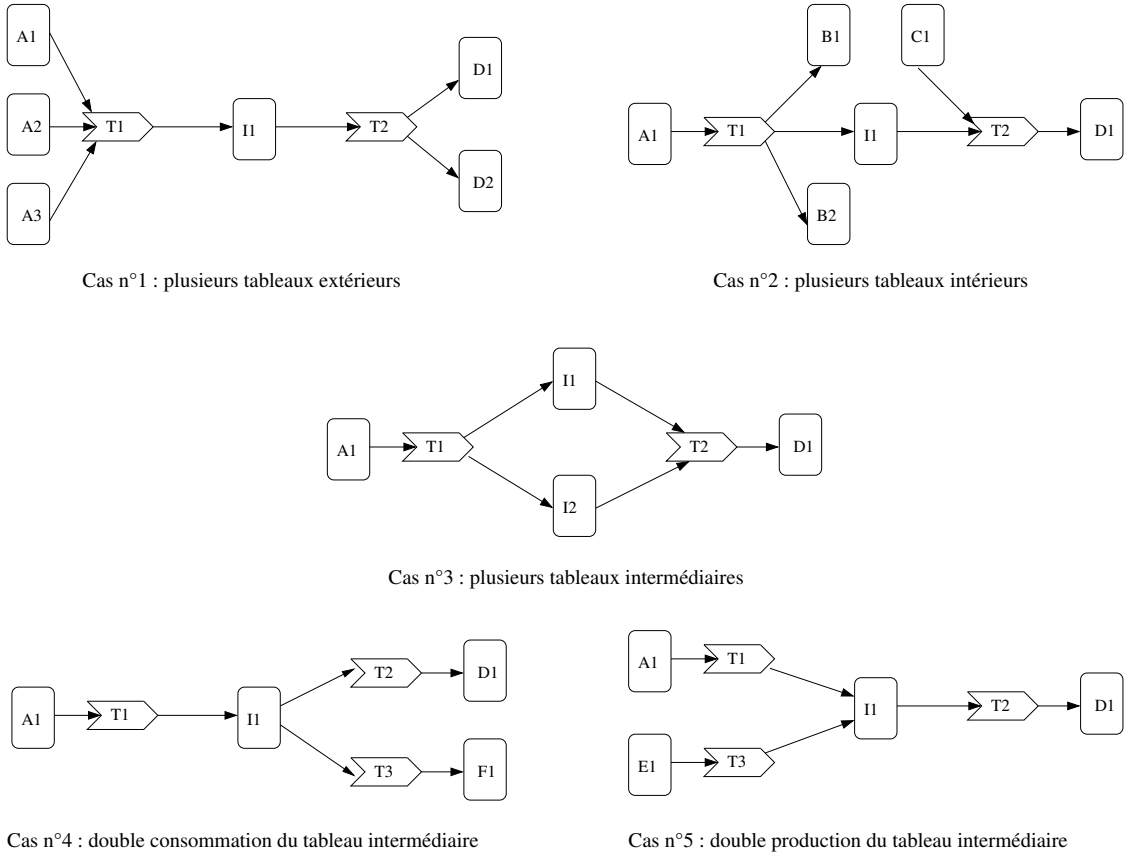


FIG. 3.6: Fusion de deux tâches avec plusieurs tableaux intermédiaires

3.2.3 Généralisation de la fusion

Nous avons détaillé le calcul de la fusion lorsqu'il y a un seul tableau en entrée et un seul tableau en sortie de chaque tâche. La généralisation de la fusion à des tâches comportant plus de tableaux est possible et se base sur les résultats que nous venons d'obtenir. Il faut distinguer cinq cas différents comme le montre la figure 3.6.

3.2.3.1 Cas n° 1

La première tâche consomme un ou plusieurs tableaux d'entrée ($A1$, $A2$ et $A3$) et symétriquement la deuxième tâche produit un ou plusieurs tableaux de sorties ($D1$ et $D2$). Il s'agit du cas le plus simple. Le calcul de la « super QD » reste identique, seul l'obtention de la hiérarchie change. Pour obtenir la hiérarchie Julien Soula propose de construire la forme ODT complète des deux tâches $T1$ et $T2$ (*cf* encadré page 55) puis d'y appliquer la « super QD » et enfin d'éclater à nouveau le résultat. Le calcul est donc analogue à celui du cas normal tel qu'il est décrit section 3.2.2.3 page 69.

Cependant cette méthode, bien que ne changeant rien du point de vue théorique, complexifie beaucoup l'implémentation. En effet, l'application de la « super QD » rend encore plus complexe la distinction des différentes tâches dans la forme ODT complète lors du ré-

éclatement. Il est en fait beaucoup plus simple d'appliquer la « super QD » sur chacune des demi-tâches et ceci de manière individuelle. Il n'y a donc pas besoin de calculer la forme ODT complète et surtout de séparer les élements de cette forme, une fois la « super QD » appliquée.

3.2.3.2 Cas n° 2

Les deux tâches peuvent avoir plusieurs tableaux en entrée et en sortie, mais elles n'en ont qu'un seul en commun. Seule diffère la gestion des tableaux intermédiaires qui ne sont pas communs aux deux tâches ($B1$,$B2$ et $C1$ du cas n° 2 de la figure 3.6). La gestion des autres tableaux se fait de manière similaire au premier cas.

Les tableaux de sortie de la premiere tâche ($B1$,$B2$) sont donc gérés de manière différente. En effet, après la fusion, ils deviennent des tableaux de sorties de la tâche supérieure tout en étant des tableaux de sorties de la première sous-tâche. Pour obtenir leurs représentations ODT après la fusion, on applique un raisonnement similaire à celui d'une tâche normale. Pour la tâche supérieure, il suffit de combiner la représentation ODT de ces tableaux avec la partie gauche de la « super QD » (et non avec la droite comme pour un tableau de sortie normale). Pour la première sous-tâche, il suffit de prendre une simple matrice identité. En effet, la première sous tâche produit un des motifs résultats de la tâche supérieure, on peut donc utiliser la matrice identité. On applique un raisonnement symétrique pour les tableaux consommés par la deuxième tâche. Le détail des calculs est présenté dans la section B.6 page 136 de l'annexe.

3.2.3.3 Cas n° 3

Les deux tâches ont plusieurs tableaux intermédiaires communs. Bien que décrivant ce cas dans sa thèse, Julien Soula ne lui apporte pas de solutions valables. La méthode que nous proposons est en fait similaire à celle du cas n° 4 qui lui n'est pas abordé par Julien Soula.

La difficulté consiste ici à exprimer les dépendances de manière correcte entre la tâche de gauche et la tâche de droite. L'utilisation de la forme complète n'est pas possible puisqu'elle rendrait non exacte la partie droite de la tâche de gauche, ce qui bloquerait l'inversion de l'ODT exact (*cf* section 3.2.2.1 page 67).

Nous proposons d'effectuer la fusion en deux étapes. Pour cela, on différencie la production de $I1$ et de $I2$, on considère alors que $I1$ est produit par la tâche $T1$, mais que $I2$ est produit par une deuxième tâche $T1$ identique à la première (*cf* figure 3.7 page 74). On applique alors deux fusions successives : la première entre la première tâche $T1$ et la tâche $T2$, la deuxième entre la deuxième tâche $T1$ et le résultat de la première fusion. Finalement, on obtient bien une seule tâche passant directement de $A1$ à $D1$ et les tableaux $I1$ et $I2$ ont disparu. En revanche la tâche obtenue comporte une hiérarchie à deux niveaux. On constate alors que si on applique n fusions successives, on a n niveaux de hiérarchies. Nous reparlerons de ce problème dans la critique de la fusion section 3.3 page 78. Autre remarque, nous avons une double exécution de $T1$: une fois pour produire $M1I2$ et une fois pour produire $M2I1$. Il peut donc y avoir des calculs redondants dont nous négligerons l'impact. Il est d'ailleurs envisageable de réutiliser une partie des résultats de l'exécution de la première $T1$ pour obtenir directement le résultat de l'exécution de la deuxième $T1$, mais si cela n'est pas toujours possible.

3.2.3.4 Cas n° 4

Le tableau intermédiaire est consommé plusieurs fois (une fois par $T2$ et une fois par $T3$).Une fusion directe($T1$-$T2$) n'est pas possible. En effet la fusion faisant disparaître le tableau intermédiaire, il ne serait pas possible de relier la tâche fusionnnée avec $T3$. La solution consiste à dédoubler le tableau intermédiaire ($I1$). On effectue ensuite une première fusion ($T1$-$T2$) qui fait disparaitre le premier tableau intermédiaire. On considère le deuxième comme un simple tableau de sorties (comme dans le cas n° 2). Puis on effectue une deuxième fusion entre le résultat de la première fusion et la tâche $T3$, ce qui fait disparaître le deuxième tableau intermédiaire. On obtient ainsi deux niveaux de hiérarchies.

La figure 3.7 montre le résultat de cette double fusion. On y voit notamment que nos trois tâches n'en forment plus qu'une seule et que le tableau intermédaire n'est jamais produit totalement. Mais on constate également qu'on a deux niveaux de hiérarchies et que la tâche $T1$ produit ses données dans deux tableaux $M1I11$ et $M1I12$. Cette double production n'est qu'apparente : $M1I11$ et $M1I12$ contiennent les mêmes données. Je les ai distingués afin de montrer que $M1I11$ est consommé par $T2$, alors que $M1I12$ sert à construire $M2I12$.

3.2.3.5 Cas n° 5

Le tableau intermédiaire est produit plusieurs fois. Ce cas n'est pas possible, c'est une conséquence immédiate de l'assignation unique.

3.2.4 Implémentation de la fusion

L'implémentation reprend point par point les étapes du calcul de la fusion telle que nous l'avons abordée dans la section 3.2.2 page 67. Pour réaliser cette implémentation, il faut être en mesure :

- de fournir une interface permettant de passer de la description ARRAY-OL d'une application à la représentation ODT d'au moins deux de ses tâches ;
- d'effectuer une suite de calcul sur les ODT ;
- de retourner la description ARRAY-OL de l'application une fois les tâches fusionnées.

De plus une telle implémentation est appelée à être utilisée dans un cadre plus large allant de la modélisation d'applications ARRAY-OL à leurs exécutions. L'implémentation proposée ici s'inscrit dans le cadre global des recherches et des développements de l'équipe WEST. Toutefois, elle n'est pas dépendante de ces derniers et peut être envisagée comme un module indépendant. Nous nous contenterons donc ici de dépeindre brièvement son fonctionnement et ses interfaces sans rentrer dans les détails.

J'ai commencé en écrivant une bibliothèque de manipulations des ODT. Cette bibliothèque propose en premier lieu une représentation des rationnels et les opérations calculatoires de base qui leurs sont accociées. En effet, une gestion personnalisée des rationels est nécessaire à cause de l'utilisation de l'infini dans les calculs. Puis j'ai écrit les classes `RationalVector` et `RationalMatrix` pour gérer les calculs sur des vecteurs ou des matrices de rationnels. Ainsi j'ai pu créer deux sous classes `OdtVector` et `OdtMatrix` contenant les primitives de bases pour la gestion des ODT sous la forme de vecteurs ou de matrices. Enfin j'ai hérité de ces classes pour créer les différents ODT : projection, segmentation, gabarit... Cette hiérarchie de classe est décrite dans le diagramme UML de la figure 3.8 page 76. La seule difficulté rencontrée à cette étape fût de bien gérer les cas limites et no-

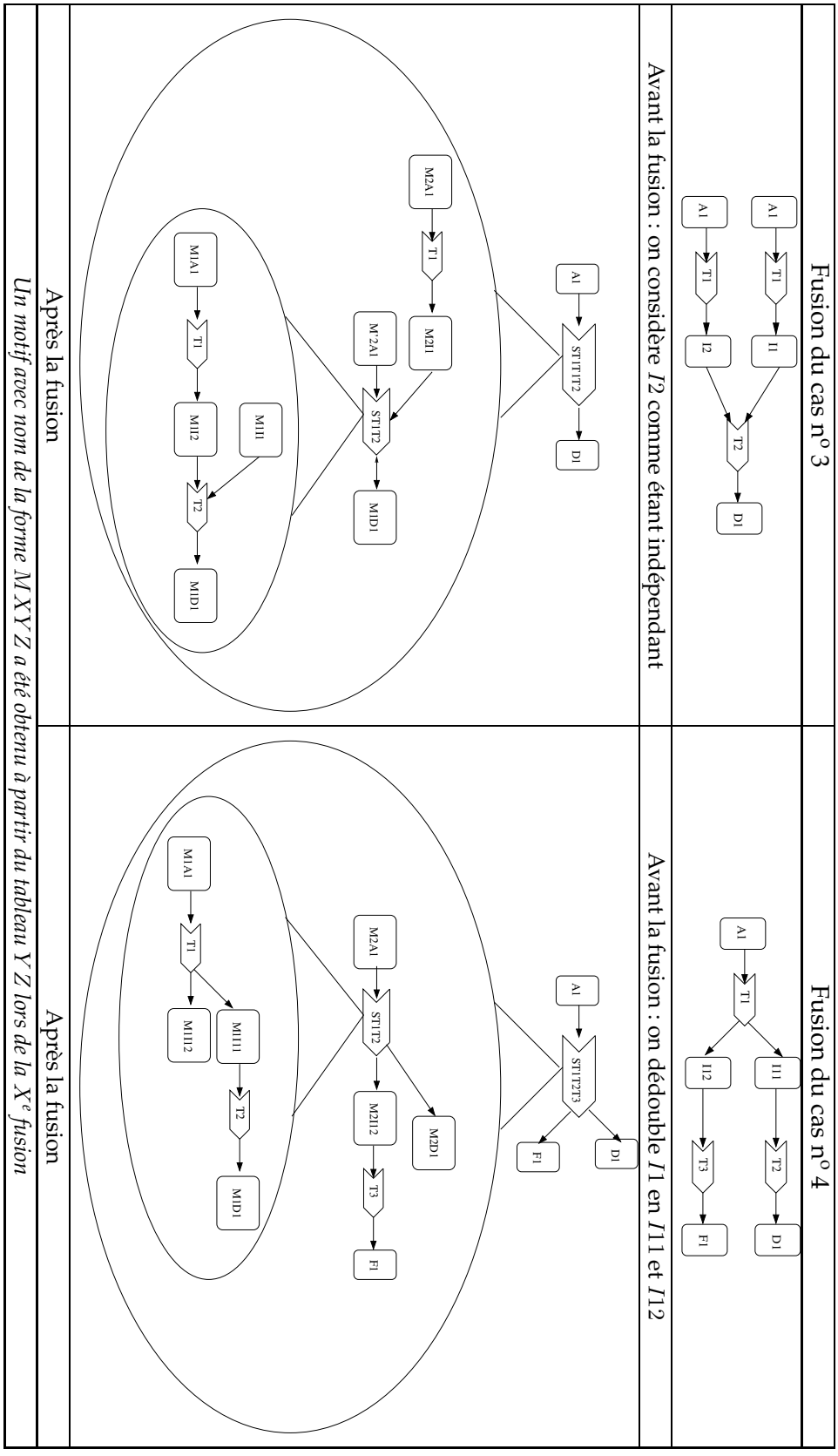


FIG. 3.7: Illustration de la fusion sur les cas n° 3 et n° 4

tamment lorsque l'une des composantes des ODT se révèlent être nulle (pas de pavage, pas d'ajustage, etc.).

Quatre classes indépendantes (*cf* figure 3.9 page 77) permettent ensuite d'effectuer les calculs de la fusion. La classe `OdtManipulations` comporte les opérations principales de la fusion telles que l'inversion des ODT exactes, le calcul de la boîte englobante ou encore les différentes simplifications. La classe `OdtTask` est utilisée pour représenter une tâche ODT complète, elle est également capable de représenter des tâches hiérarchiques. Ainsi la méthode fusion de la classe `OdtFusion` prend deux `OdtTask` comme paramètres et retourne une `OdtTask` hiérarchique. La classe `OdtHalfTask` est, quant à elle, utilisée pour représenter une demi-tâche et y appliquer les résultats de la fusion dans le cas n° 2 de la fusion généralisée (*cf* section 3.2.3.2 page 72).

Ces classes fournissent donc les bases nécessaires au calcul de la fusion, il faut donc maintenant concevoir une API permettant de passer de la représentation d'une application ARRAY-OL à sa représentation ODT. En effet comme nous l'avons vu dans la section 2.4.2.3 page 54, il n'y a pas d'équivalence des représentations ARRAY-OL et ODT. Cette API est donc indispensable à l'utilisation de la fusion.

Pour ne pas avoir à gérer toute une application, ce qui serait inutile puisque seules les tâches (les modèles locaux) sont intéressantes, l'API autorise simplement la création de tâches via la classe `OdtApiTask`. Ces classes sont utilisées comme paramètres de la méthode fusion de la classe `OdtApiFusion` qui retourne à son tour une `OdtApiTask` hiérarchique. Le principe est donc similaire à celui de la gestion des tâches ODT. Mais cette fois-ci, une `OdtApiTask`, contrairement à une `OdtTask`, contient toutes les informations nécessaires à son insertion dans une description globale d'application. Si on regarde plus en détails le fonctionnement de cette API, on peut schématiser le déroulement d'une fusion de la manière suivante :

- **Création des deux tâches opérandes :** on crée d'abord deux objets `OdtApiTask` vides. Puis on se sert des méthodes `addInputDependence` et `addOutputDependence` pour ajouter les demi-tâches une par une. En effet le nombre de demi-tâches n'étant pas toujours le même, il n'est pas possible de passer par le constructeur. On en profite pour associer un identifiant unique (`OdtApiConnectionPoint`) au motif et au tableau de chaque demi-tâche.
- **Calcul de la fusion :** on appelle la méthode fusion de la classe `OdtApiFusion` en passant nos deux `OdtApiTask` comme paramètre. Cette méthode recherche d'abord les tableaux intermédiaires communs grâce aux identifiants uniques. Puis elle calcule la représentation ODT des deux tâches et finalement elle appelle la méthode fusion de la classe `OdtFusion`.
- **Retour du résultat :** avant de retourner un résultat il faut convertir le résultat `OdtTask` que l'on vient d'obtenir en `OdtApiTask`. Le principal travail est d'associer à chaque nouveau tableau et à chaque nouveau motif un nouvel identifiant. Ainsi, le motif d'entrée de la tâche supérieure aura le même identifiant que le tableau d'entrée de la sous-tâche ; le tableau d'entrée de la tâche supérieure aura le même identifiant que le tableau d'entrée de la tâche de gauche avant la fusion et ainsi de suite… Cette technique est utilisée pour qu'on puisse relier la tâche fusionnée correctement dans l'application. Enfin on retourne une `OdtApiTask` contenant le résultat de la fusion.

Cette API fournit donc une interface « basique » pour la fusion de tâches ARRAY-OL. Son utilisation nécessitera généralement l'établissement d'une passerelle faisant office de convertisseur entre les `OdtApiTask` et le modèle de données de l'application appelante.

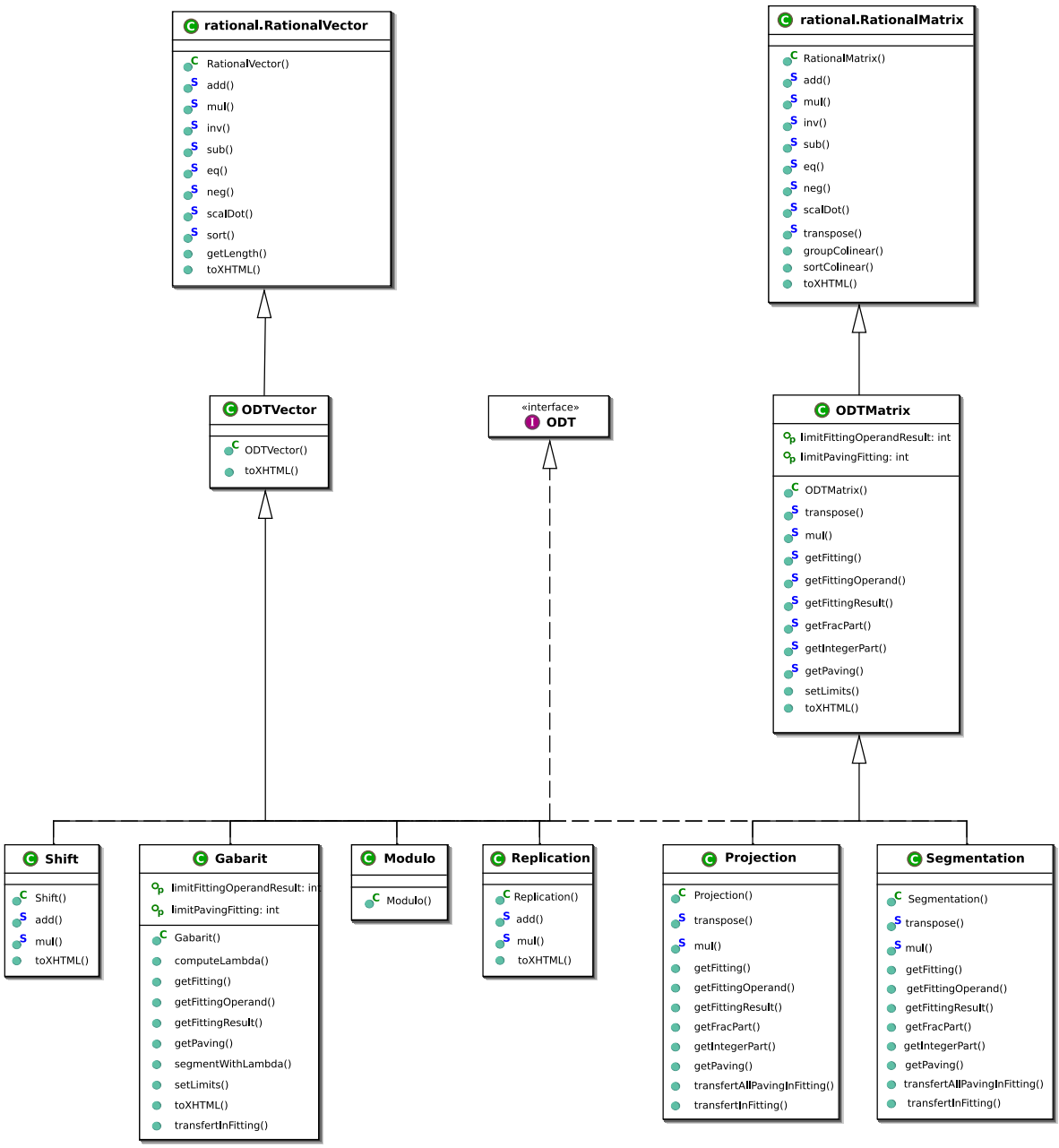


FIG. 3.8: Hiérarchie des ODT

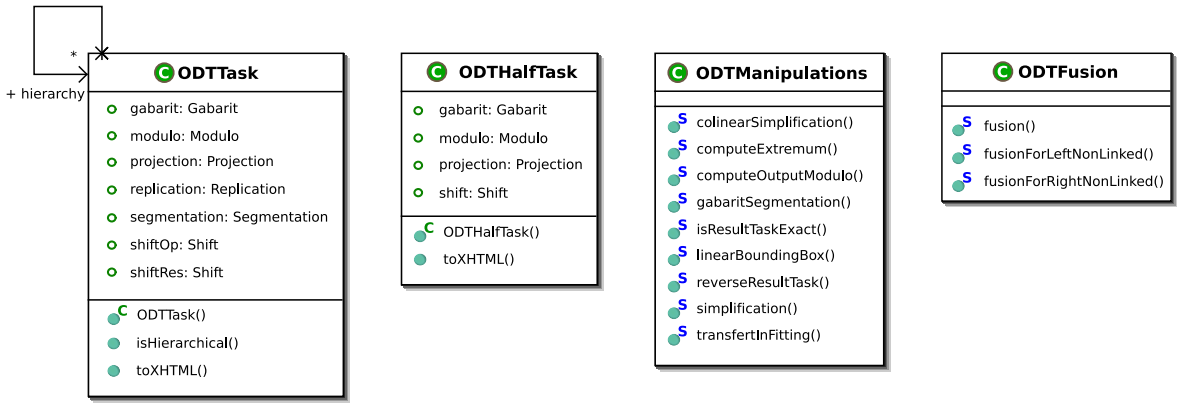


FIG. 3.9: Classes de manipulation des ODT

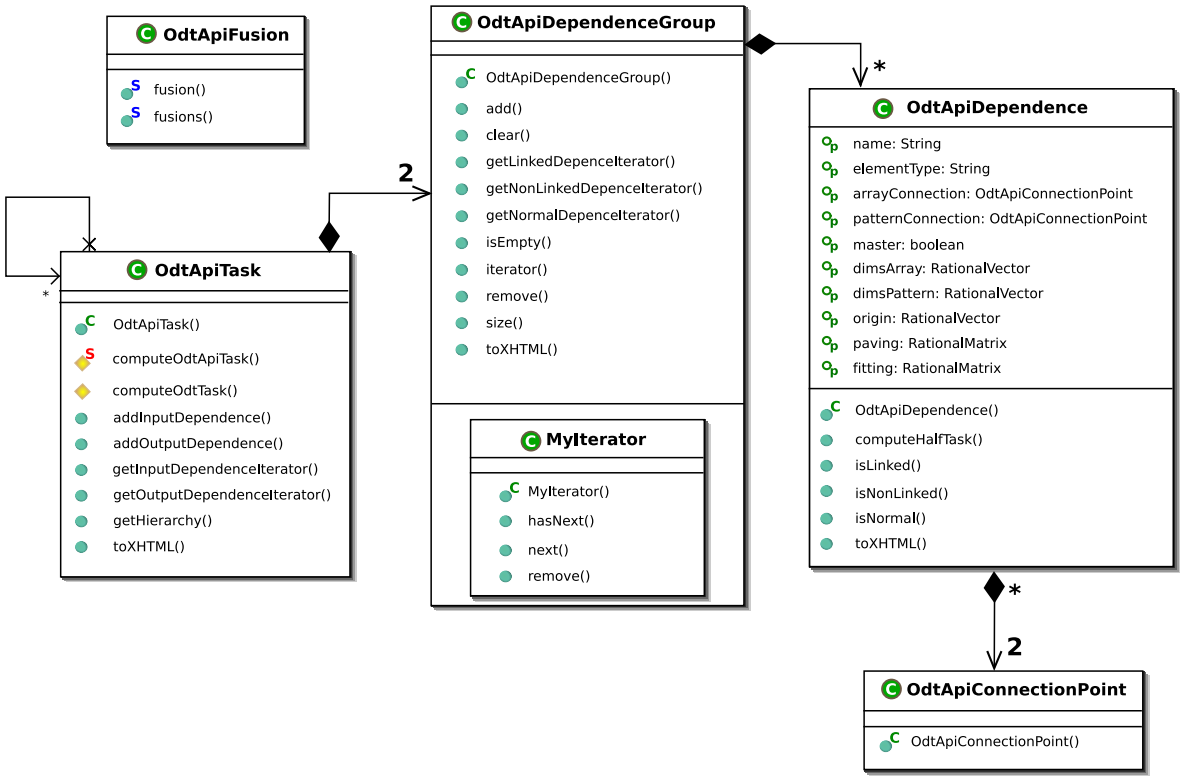


FIG. 3.10: L'API de manipulation de la fusion

3.2.5 Conclusion

Nous venons de voir le fonctionnement de la fusion de deux tâches, d'abord en étudiant l'aspect théorique d'un cas simple, puis en généralisant les calculs à l'ensemble des cas et enfin en proposant une implémentation. Il est apparu que la fusion ne comportait pas que des avantages : les sous-tâches qu'elle génére ne sont pas nécessairement exactes, la succession de plusieurs fusions entraîne la création de tâches hiérarchiques à multiples niveaux. Ce ne sont là que les défauts les plus visibles, il en existe d'autres qui se révèlent être plus génants. Nous allons donc proposer dans la prochaine section , une suite de correctifs à la fusion qui donneront naissance à de nouvelles optimisations des applications ARRAY-OL.

3.3 Critique de la fusion

La fusion ne comporte pas que des avantages, on peut en fait distinguer quatre inconvénients principaux.

Inéxactitude des résultats Le premier inconvénient de la fusion apparaît à la lecture de la forme ODT des résultats (*cf* section 3.2.2.3 page 69). En effet, la première sous-tâche n'est pas nécessairement exacte puisqu'elle comporte un *shift* résultat.

Pour qu'une tâche soit exacte, il faut également que sa segmentation respecte un certain nombre de conditions (*cf* section 1.3.2.3 page 26). Or nous n'avons rien prouvé concernant les segmentations de la tâche supérieure et de la première sous-tâche. La segmentation de la deuxième sous-tâche, quant à elle, est forcément exacte puisqu'elle est égale à la matrice identité. L'étude de l'éxactitude de ces deux segmentations dépend également des calculs intermédiaires de la fusion. Il résulte que la segmentation de la tâche supérieure est exacte, mais que ce n'est pas toujours le cas pour celle de la première sous-tâche. Cependant la non exactitude de cette segmentation ne constitue pas une grosse entorse au principe d'ARRAY-OL puisque les bornes de pavage de cette sous-tâche sont déjà connues grâce à la fusion. Cette dernière impose en effet que les n bornes de pavages soient égales à la taille des n premières dimensions du tableau d'entrée ($A4$).

Création de hiérarchies abyssales Comme nous l'avons vu à la section 3.2.3.3 page 72, une succession de fusions augmente la profondeur de la hiérarchie. Ainsi si l'on effectue n fusions successives sur des tâches situées à un même niveau alors la profondeur maximum de la hiérarchie est de n. On parle alors de hiérarchies abyssales. Ces hiérarchies complexifient la compréhension de l'application. Il serait donc intéressant de trouver une nouvelle transformation qui permettrait de niveler les tâches. Ainsi on pourrait fusionner successivement n tâches, puis remonter les niveaux de hiérarchies les plus bas pour obtenir une simple tâche hiérarchique mais qui permettrait de passer directement du tableau opérande de la première tâche au tableau résultat de la dernière. De plus l'exécution de cette tâche serait totalement data-parallèle. Nous verrons ultérieurement comment obtenir une telle transformation.

Réduction limitée de l'occupation mémoire Un autre inconvénient concerne la réduction de l'occupation mémoire. Le gain de place est dû à la suppression du tableau intermédiaire qui est remplacé par le tableau $A5$ (*cf* figure 3.1 page 64 et figure 3.2 page 64). Les éléments de $A5$ sont uniquement produits par $T1$. Or $T1$ produit ses éléments en respectant le système

de coordonnées de $A2$, ce qui implique donc que $A5$ respecte la forme de $A2$ et par la même que $A5$ a comme taille la boîte englobante du macro-motif produit par $T1$. Et de ce fait, la taille de $A5$ peut rester relativement importante. La figure 3.11 présente la situation dans un cas favorable tandis que la figure 3.12 montre un cas défavorable.

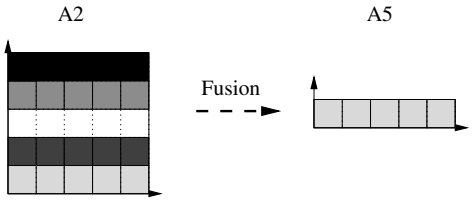


FIG. 3.11: cas favorable

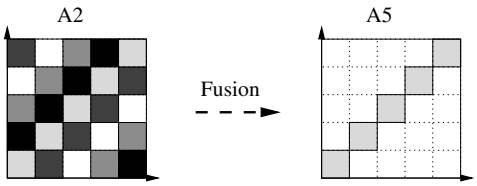


FIG. 3.12: cas défavorable

Aucune solution n'a été mise en œuvre pour résoudre ce problème , bien que Julien Soula ait proposé d'utiliser une fonction de réindexage.

Apparition de recalcul Enfin, la fusion peut également provoquer l'apparition de recalculs. L'étude de ce problème et les conséquences qui en découlent font l'objet de la section suivante.

3.4 Apparition de recalculs

3.4.1 Mise en évidence des recalculs

Une des conséquences les moins visibles de la fusion est l'apparition de recalculs dans certains cas particuliers. Ces recalculs interviennent au niveau de la première sous-tâche de la hiérarchie. Pour mieux comprendre ce qui se passe, examinons l'exécution d'une tâche hiérarchique issue d'une fusion :

- la tâche du haut consomme un macro-motif constitué d'un agglomérat de motifs de la première tâche avant la fusion (*cf* section 3.2.1 page 63) ;
- ce macro-motif devient le tableau d'entrée de la première sous-tâche ;
- or cette première sous-tâche effectue le même traitement que la première tâche avant la fusion, elle consomme donc les mêmes motifs (*cf* section 3.2.1 page 63) ;
- le macro-motif est donc découpé en motifs originaux par cette sous-tâche ;
- le recalcul intervient lorsque deux macro-motifs partagent des motifs originaux en commun, ce qui fait que deux exécutions de la sous-tâche vont consommer les mêmes motifs originaux.

La figure 3.13 illustre la présence d'un recalcul. Si l'on observe le tableau d'entrée de la tâche supérieure après la fusion, on peut voir que deux macro-motifs successifs ont trois éléments communs. Ces éléments constituent en fait chacun des motifs originaux de la première tâche ($T1$) avant la fusion. Ils vont donc être consommé un à un par la première sous-tâche puisqu'il s'agit de la même tâche $T1$. Mais comme ils sont présents dans deux macro-motifs, ils sont consommés deux fois chacun ! Il y a donc recalcul.

Si on étudie l'exécution complète de la tâche fusionnée, on comprend alors que la majorité des éléments ne sont pas calculés deux fois par $T1$ mais bel et bien quatre fois. En effet comme le montre la figure 3.14 page 81 tous les points d'un macro-motif, à l'exception des

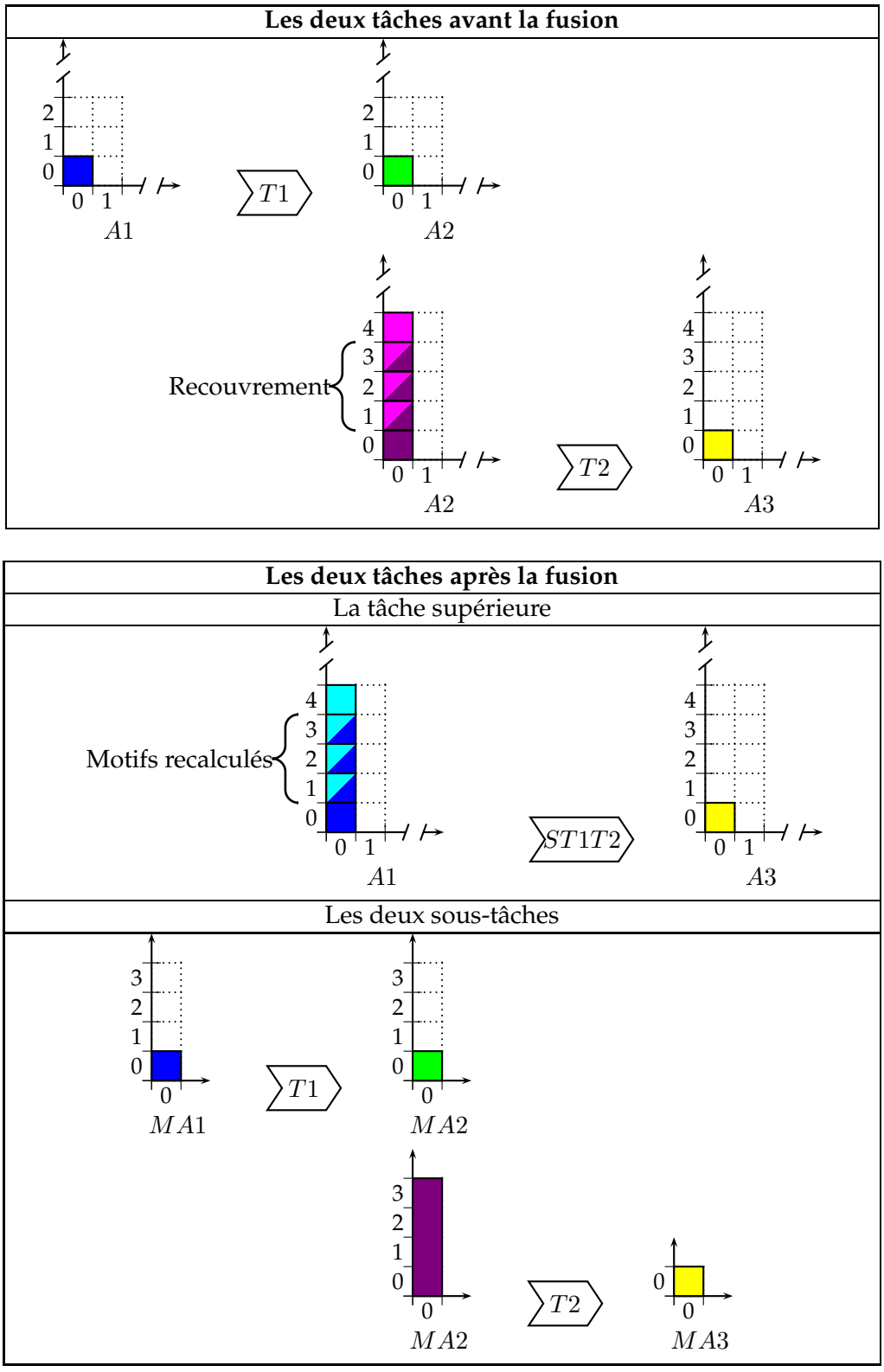


FIG. 3.13: Exemple de recalcul

extrêmes, appartiennent en fait à trois autres macro-motifs. Ainsi le temps d'exécution de la tâche $T1$ est quadruplé. On peut d'ailleurs constater que ce recalcul est le fruit de la fusion, il n'était pas présent avant comme le montre la figure 3.13 .

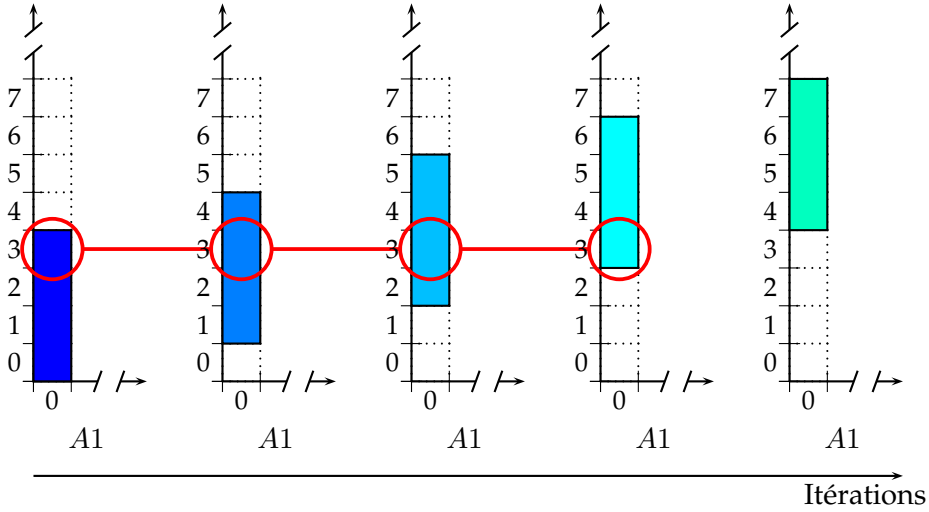


FIG. 3.14: Nombre de motifs communs à un même élément

3.4.2 Origine des recalculs

Après une fusion, des recalculs peuvent donc intervenir au niveau de la première sous-tâche ($T1$) d'une tâche hiérarchique. Ces recalculs sont toujours dus aux manipulations qui sont faites sur le tableau opérande ($A1$) de la tâche supérieure. Ainsi, il y a un recalcul lorsque différents macro-motifs du tableau opérande partagent des élements communs et que ces éléments constituent au moins un motif opérande de la première tâche avant la fusion. Le partage d'éléments entre macro-motifs peut se produire dans deux cas distincts :

- Premièrement, lorsque les macro-motifs de $A1$ se chevauchent ; c'est l'exemple de la figure 3.13 .
- Deuxièmement, lorsque $A1$ est torique sur au moins une de ses dimensions. Les premiers motifs sont recalculés une fois que l'on a fait le « tour » du tableau.

Bien qu'ayant les mêmes effets, ces deux cas n'ont ni la même cause, ni la même gravité. Le premier cas est directement la conséquence de la fusion puisque le recalcul n'apparait pas dans les deux tâches originelles (*cf* figure 3.13), alors que dans le deuxième cas il est impératif que le recalcul soit présent avant la fusion puisque cette dernière ne rend pas les dimensions toriques[2]. De plus si on compare leurs degrés de recalcul, on constate que le chevauchement des macro-motifs quadruple le temps d'exécution de la première sous-tâche alors que l'utilisation du modulo nécessite de parcourir quatre fois le tableau opérande pour obtenir un tel résultat.

Les recalculs par utilisation du modulo ne sont donc pas une conséquence directe de la fusion, il est en outre possible de réduire leurs effets au moment de la modélisation de

[2]La fusion n'introduit pas l'utilisation de modulo, il faut que le modulo soit déjà présent dans la consommation d'un des tableaux opérandes d'une des tâches originelles avant la fusion.

l'application. Nous essaieront donc de résoudre ici uniquement les recalculs provoqués par le chevauchement des macro-motifs.

La cause de ces chevauchements est relativement simple à trouver. La fusion permet de passer directement des tableaux $A1$ à $A3$ sans passer par $A2$. Pour cela, elle agglomère suffisament de motifs de $A1$ pour être en mesure de construire au moins un motif de $A3$. Cet agglomérat est donc le macro-motif. Or s'il y a chevauchement des macro-motifs, c'est que pour construire un motif de $A3$, on utilise une partie des données ayant servi à construire un autre motif de $A3$. Il faut donc qu'il y ait également un chevauchement des données nécessaires à la construction de $A3$. En effet comme le montre la figure 3.13 page 80, si les motifs opérandes de $T2$ se recouvrent alors les macro-motifs se chevaucheront[3].

3.4.3 Réduction des recalculs

Pour réduire les recalculs, on propose simplement de limiter les chevauchements. Pour cela, on augmente le vecteur de pavage de la dimension sur laquelle se fait le chevauchement et on agrandit la taille du macro-motif. La figure 3.15 montre un agrandissement du macro-motif dans le cadre de notre exemple des figures 3.13 page 80 et 3.14 . Dans ce cas précis la taille du macro-motif et le vecteur de pavage ont été augmentés de 1. Ainsi comme le montre la figure, un élément du tableau appartient soit à trois macro-motifs différents, soit seulement à deux. Le nombre d'exécution de $T1$ est donc multiplié par deux et demi au lieu de quatre. De plus si la taille du macro-motif est encore augmentée, le nombre de recalcul diminuera. Ainsi plus la taille du macro-motif est grande plus le nombre de recalculs est petit.

Cependant cette technique a un certain nombre d'impacts sur notre tâche hiérarchique. Tout d'abord, elle modifie la taille du macro-motif résultat produit par la tâche supérieure. Pour comprendre les enjeux, étudions le changement de taille du macro-motif dans le cadre de notre exemple. La première sous-tâche ($T1$) va consommer les cinq éléments pour en produire cinq autres, la deuxième sous-tâche, quant à elle, va consommer les quatres premiers éléments (numérotés de 0 à 4) pour produire un motif résultat final. Mais elle consommera également les éléments numérotés de (1 à 5) pour produire un deuxième résultat final. En effet, notre nouveau macro-motif contient deux macro-motifs originaux (celui constitué par les points numérotés de 0 à 4 et celui constitué par les points numérotés de 1 à 5). Il est donc logique qu'une fois agrandit, le macro-motif serve à produire les deux résultats.

Le deuxième impact de notre technique est un agrandissement de la taille de tous les tableaux de la sous-tâche. On diminue donc la réduction de l'occupation mémoire offerte par la fusion. Toutefois cette diminution reste marginale face à la réduction du recalcul si on n'agrandit pas le macro-motif de façon trop importante.

On peut également trouver une autre utilité à cette technique ; il est en effet possible d'utiliser le redimensionnement du macro-motif pour définir la granularité d'une hiérarchie qu'elle soit issue ou non d'une fusion ou qu'elle comporte ou non des recalculs. Ainsi il est possible de choisir pour les itérations de la tâche supérieure le nombre de données consommées et donc par la même de choisir le nombre de données produites. À l'instar de la fusion, cette technique propose de transformer une application ARRAY-OL, nous l'appelons changement de pavage *(change paving)* car il faut changer les vecteurs de pavage afin de l'appliquer.

[3]On peut également constater sur cette figure que le chevauchement avant la fusion n'entraîne pas de recalculs puisque la tâche $T2$ consomme des motifs de taille quatre qui sont tous différents les uns des autres

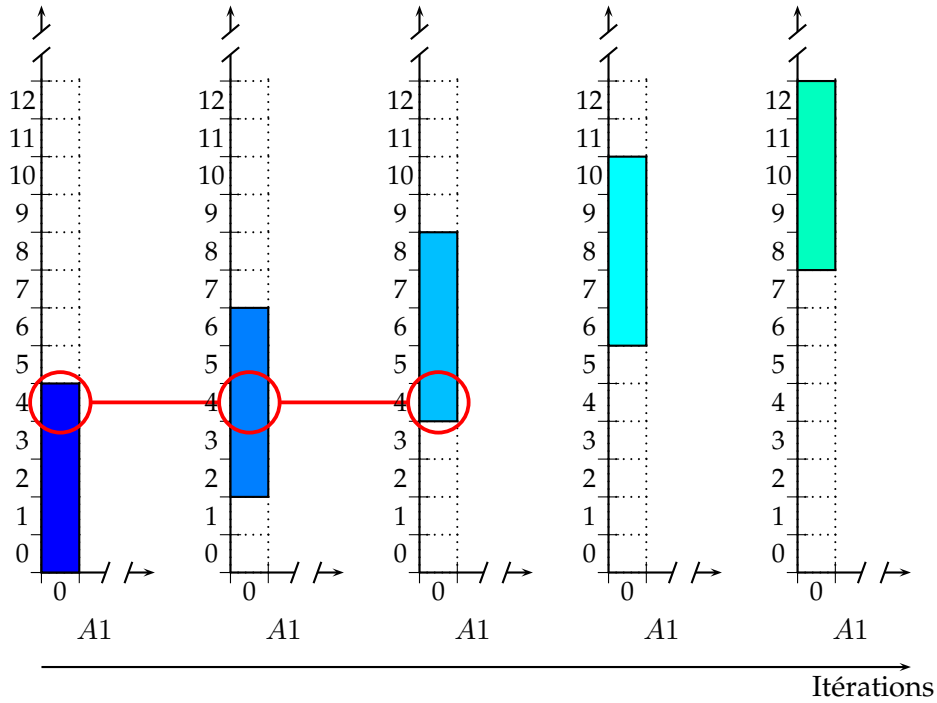


FIG. 3.15: Effet de l'agrandissement du macro-motif sur les recalculs

Notons tout d'abord que le changement de pavage est déja présent dans la thèse de Julien Soula. Mais la technique qu'il propose pour agrandir le macro-motif repose sur l'emploi de calculs intermédiaires de la fusion, ce qui empêche son utilisation en dehors du cadre strict d'un résultat de fusion dont on aurait gardé les étapes intermédiaires. Nous proposons donc ici une nouvelle méthode reposant sur une démarche originale.

Le changement de pavage consiste donc à augmenter la taille d'un des vecteurs de pavage et à augmenter la taille du macro-motif. Il est bien sûr obligatoire de faire correspondre l'agrandissement du macro-motif et la modification du vecteur pavage afin de ne pas modifier les résultats produits.

3.4.4 Changement de pavage par ajout de dimensions

Notre première idée pour mettre en œuvre le changement de pavage fut de proposer l'ajout d'une dimension au macro-motif. Cette dimension sert de « rangement » pour mettre côte à côte des macro-motifs originaux. Sur l'exemple de la figure 3.16 , on décide d'utiliser le changement de pavage pour diviser par deux le nombre d'itérations de la tâche.

Notre but est donc de calculer en une seule et même itération deux macro-motifs résultats (en jaune clair et foncé sur la figure). Par conséquent il nous faut également consommer deux macro-motifs opérandes à la fois (en bleu clair et foncé). On décide alors d'agrandir le macro-motif en lui ajoutant une dimension de taille deux pour « loger » les deux macro-motifs originels. Ainsi le nouveau macro-motif opérande est de taille $2 \times 3 \times 3$ et celui résultat est de taille 2×1. Il nous faut désormais modifier le pavage et l'ajustage opérande et résultat pour consommer et produire correctement ces nouveaux macro-motifs.

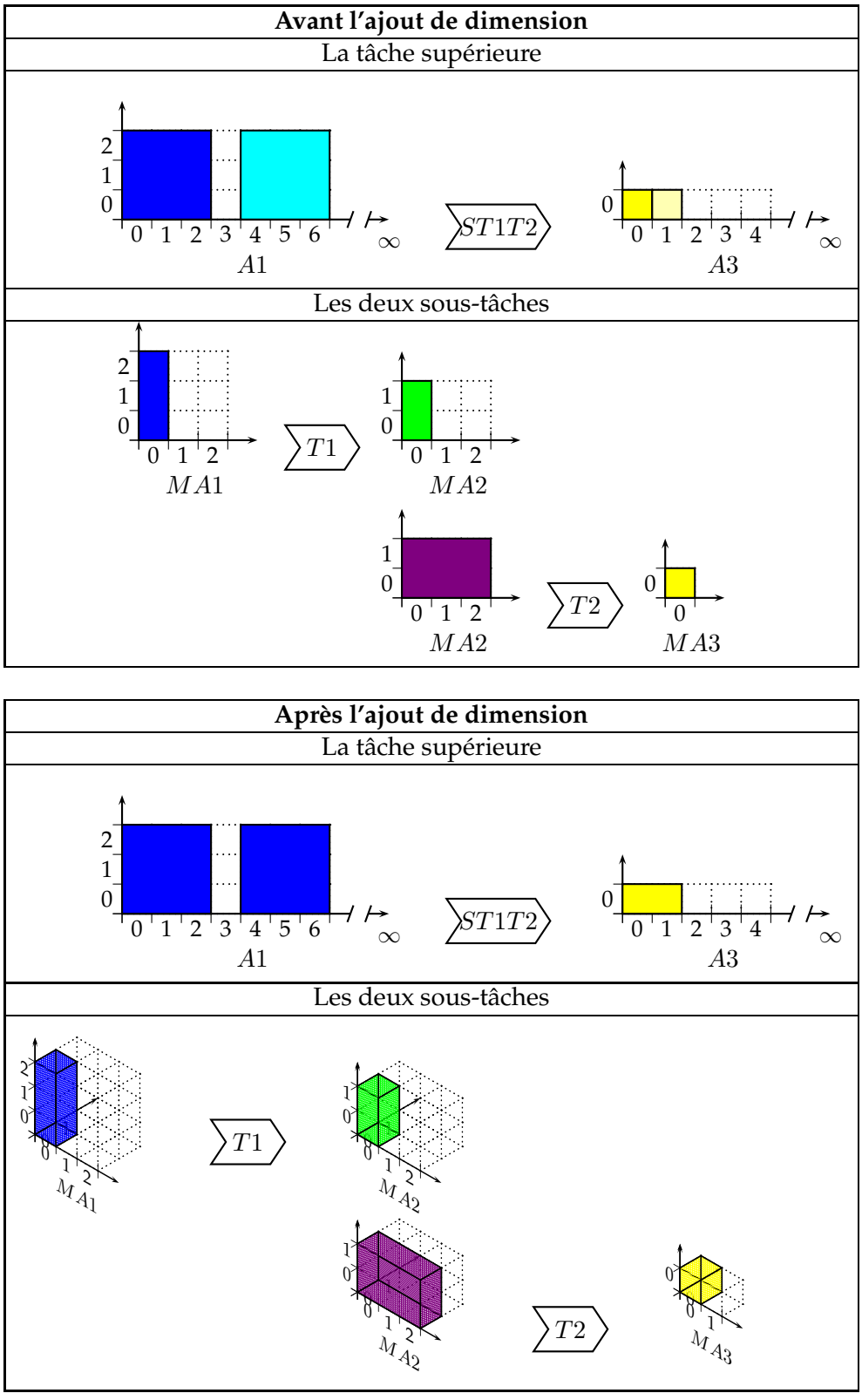


FIG. 3.16: Changement de pavage par ajout de dimensions

L'opération est relativement simple : il suffit de faire passer le vecteur de pavages dans l'ajustage. Ainsi les « deuxièmes » macro-motifs (de couleur claire sur la figure) qui n'étaient atteignables que par le pavage le sont désormais par l'ajustage. De plus, pour que le pavage ne les prenne plus en compte, on double la taille du vecteur de pavage orginel, ainsi on passe directement du « premier » au « troisième » macro-motif. Sur notre exemple le pavage est doublé et passe de (4) à (8) du coté opérande et de (1) à (2) du coté résultat.

Nous venons d'effectuer la modification de la tâche supérieure, il faut maintenant modifier les deux sous-tâches. La première consomme un tableau de taille $2 \times 3 \times 3$, donc le tableau qu'elle produit voit également sa taille doublée, il passe de 3×2 à $2 \times 3 \times 2$. En appliquant le même raisonnement on en déduit que tous les tableaux de la sous-tâche gagnent une dimension. Pour atteindre cette nouvelle dimension il suffit de rajouter un vecteur de pavage de forme $\begin{pmatrix} 1 \\ 0 \\ \dots \end{pmatrix}$ à chaque demi-tâche concernée.

Étudions désormais l'ajout d'une dimension d'un point de vue théorique. La première étape est de choisir un vecteur de pavage puis de choisir le coefficient d'agrandissement du macro-motif. Dans l'exemple précédent, nous avons doublé la taille du macro-motif ; mais il est tout à fait possible de tripler ou de quadrupler cette taille, la dimension qu'on rajoute est simplement plus grande. Il faut cependant que le coefficient d'agrandissement soit un diviseur de la borne d'itération du vecteur de pavage. En effet, si on choisit un vecteur ayant une borne de cinq, il n'est pas possible de multiplier par deux la taille du macro-motif, puisque cela impliquerait qu'un macro-motif originel reste seul. Une fois ces données choisies, il suffit de faire une copie du vecteur de pavage dans l'ajustage et de multiplier ce dernier par le coeffcient. On divise alors la borne de pavage par le coefficient. La création des sous-tâches ne comporte aucune difficulté, il faut juste ajouter à chaque tableau la nouvelle dimension de taille égale au coefficient. Il faut également ajouter à chaque tableau un nouveau vecteur de pavage à l'image de celui décrit dans notre exemple. Soit l'expression ODT d'une tâche ARRAY-OL :

$$(M)_{\mathbf{M}} . (S)_{\mathbf{S}} . \left| P_{op} \quad F_{op} \quad 0 \right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res} \quad 0 \quad F_{res}}} \ .$$

Considérons cette expression comme celle de la tâche supérieure d'une hiérarchie et appliquons y notre technique d'ajout de dimensions : soit i l'indice d'un vecteur de pavage, on chosit n le facteur d'agrandissement tel que $Q_i \mod n = 0$. On obtient alors :

$$(M)_{\mathbf{M}} . (S)_{\mathbf{S}} . \left| P_{op,1} \quad \dots \quad n \times P_{op,i} \quad \dots \quad P_{op,m} \quad P_{op,i} \quad F_{op} \quad 0 \quad 0 \right|$$
$$. \begin{pmatrix} Q_1 \\ \vdots \\ Q_i/n \\ \vdots \\ Q_m \\ n \\ D_{op} \\ n \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{P_{res,1} \quad \dots \quad n \times P_{res,i} \quad \dots \quad P_{res,m} \quad 0 \quad 0 \quad P_{res,i} \quad F_{res}}} \ .$$

Faisons de même, en considérant l'expression comme celle d'une sous-tâche de la hiérarchie. Après application des modifications on a alors :

$$\begin{pmatrix} n \\ M \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} 0 \\ S \end{pmatrix}_{\mathbf{S}} . \left|\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & P_{op} & F_{op} & 0 \end{matrix}\right| . \begin{pmatrix} n \\ Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & P_{res} & 0 & F_{res} \end{matrix}} \quad .$$

Nous venons de détailler le changement de pavage par ajout de dimensions, étudions maintenant son impact sur la réduction du recalcul en reprenant notre exemple de la figure 3.13 page 80. Nous sommes hélas forcés de constater que l'ajout d'une dimension au macro-motif ne changerait rien au recalcul. En effet, un macro-motif de taille double contiendrait deux macro-motifs originels et donc il contiendrait deux fois les points qui font l'objet du recouvrement ; le recalcul subsisterait. Il faut donc proposer un nouveau changement de pavage en agrandissant linéairement la taille du macro-motif et en ajoutant uniquement des éléments ne s'y trouvant pas déjà.

Le changement de pavage par ajout de dimensions est-il inutile ? En fait non, car si on considère l'exemple de la figure 3.16 page 84, on constate alors qu'un agrandissement linéaire du macro-motif engloberait les points de coordonnées $\binom{3}{0}$, $\binom{3}{1}$, $\binom{3}{2}$. Ces points ne faisant partie d'aucun macro-motif leur traitement poserait immanquablement des problèmes. Nous allons donc voir dans la section suivante dans quel cas le changement de pavage par agrandissement linéaire du macro-motif manipule exactement le bon nombre de points.

3.4.5 Changement de pavage par agrandissement linéaire : partie opérande

Dans un premier temps, nous allons mettre en équation la présence d'un recalcul par chevauchement dans une tâche hiérarchique. Puis nous étudierons le changement de pavage par agrandissement linéaire au niveau de la partie opérande de la tâche supérieure.

3.4.5.1 Mise en équation d'un recouvrement

Nous nous basons pour cela sur la représentation ODT de la tâche supérieure. Nous distinguons dans cette représentation les dimensions de macro-ajustage et d'ajustage spatial [4]. On obtient :

$$(M_{op})_{\mathbf{M}} . (S_{op})_{\mathbf{S}} . \left|\begin{matrix} P_{op} & \mathcal{M}F_{op} & F_{op} & 0 & 0 \end{matrix}\right| . \begin{pmatrix} Q \\ \mathcal{M}D_{op} \\ D_{op} \\ \mathcal{M}D_{res} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} P_{res} & 0 & 0 & \mathcal{M}F_{res} & F_{res} \end{matrix}} \quad .$$

Pour qu'il y ait recouvrement donc recalcul, il faut que chaque itération de pavage partage un motif de la sous-tâche avec une autre itération de pavage. Il nous suffit en fait de prouver qu'elles partagent seulement les origines en n'utilisant que le pavage et le macro-ajustage, l'ajustage spatial servant à recréer le reste du motif. On désigne alors par i la dimension de pavage sur laquelle se trouve le recouvrement et par k la dimension de macro-ajustage permettant d'atteindre les origines. On peut alors commencer à écrire les conditions

[4] Pour mémoire, le macro-ajustage devient le pavage dans la sous-tâche, alors que l'ajustage spatial devient l'ajustage.

menant à l'établissement de l'équation du recalcul :

$$\exists i \in \mathbb{N}, \exists k \in \mathbb{N}, \forall \overrightarrow{q}, \overrightarrow{0} \leq \overrightarrow{q} < \overrightarrow{Q} - \overrightarrow{1}, \exists \overrightarrow{q'}, \overrightarrow{0} \leq \overrightarrow{q'} < \overrightarrow{Q}, q'_i = q_i + 1 \text{ et } \forall j \neq i, q'_j = q_j,$$

Cette première ligne implique qu'on ne prend en compte le recouvrement qu'au niveau des itérations de pavage successives sur une et une seule dimension.

$$\exists \overrightarrow{d}, \overrightarrow{0} \leq \overrightarrow{d} < \overrightarrow{\mathcal{M}D}, \exists \overrightarrow{d'}, \overrightarrow{0} \leq \overrightarrow{d'} < \overrightarrow{\mathcal{M}D}, \forall j \neq k, d'_j = d_j,$$

Cette deuxième ligne implique qu'on ne prend en compte que les origines atteignables par le même vecteur d'ajustage ce qui ne constitue pas en soit une limitation importante. On peut désormais écrire la condition principale du recouvrement :

$$\begin{aligned} \overrightarrow{O} + \left|P_{op} \quad \mathcal{M}F_{op}\right| . \begin{pmatrix} q \\ d \end{pmatrix} &\equiv \overrightarrow{O} + \left|P_{op} \quad \mathcal{M}F_{op}\right| . \begin{pmatrix} q' \\ d' \end{pmatrix} \mod \overrightarrow{M} \\ \overrightarrow{P_{op,i}} \times q_i + \overrightarrow{\mathcal{M}F_{op,k}} \times d_k &\equiv \overrightarrow{P_{op,i}} \times q'_i + \overrightarrow{\mathcal{M}F_{op,k}} \times d'_k \mod \overrightarrow{M} \end{aligned}$$

Or on a posé que $q'_i = q_i + 1$, on obtient donc :

$$\begin{aligned} \overrightarrow{\mathcal{M}F_{op,k}} \times d_k &\equiv \overrightarrow{P_{op,i}} + \overrightarrow{\mathcal{M}F_{op,k}} \times d'_k \mod \overrightarrow{M} \\ \overrightarrow{P_{op,i}} - \overrightarrow{\mathcal{M}F_{op,k}} \times (d_k - d'_k) &\equiv 0 \mod \overrightarrow{M} \end{aligned}$$

On pose $\Delta = d_k - d'_k$, on a donc $-D_k < \Delta < D_k$ et

$$\overrightarrow{P_{op,i}} - \overrightarrow{\mathcal{M}F_{op,k}} \times \Delta \equiv 0 \mod \overrightarrow{M} \tag{3.2}$$

On obtient donc une équation permettant d'établir s'il y a recalcul ou non. Cette équation est d'ailleurs facilement compréhensible d'un point de vue ARRAY-OL puisqu'elle signifie simplement que pour qu'il y ait recalcul il faut qu'une origine de motif originel soit atteignable par une dimension de macro-ajustage dans un premier macro-motif $\overrightarrow{\mathcal{M}F_{op,k}} \times d_k$ et que cette même origine soit également atteignable par la même dimension de macro-ajustage $\overrightarrow{\mathcal{M}F_{op,k}} \times d'_k$ après une itération de pavage sur le vecteur $\overrightarrow{P_{op,i}}$.

3.4.5.2 Validité du changement de pavage par agrandissement linéaire

Nous étudions ici la partie consommation de la tâche supérieure d'une tâche hiérarchique sur laquelle on a appliqué le changement de pavage par agrandissement linéaire. Pour simplifier la mise en application du changement de pavage on considère qu'il faut que l'ensemble des points consommés par l'exécution de la tâche soit identique à l'ensemble des points consommés avant l'application du changement de pavage.

Soit U l'ensemble des points consommés par une tâche, on peut alors facilement écrire à l'aide des ODT que :

$$U = \left\{ \overrightarrow{u} \middle| \overrightarrow{u} \equiv \left(\overrightarrow{O} + \left|P_{op} \quad F_{op}\right| . \begin{pmatrix} q \\ d \end{pmatrix} \right) \mod \overrightarrow{M_{op}}, \overrightarrow{0} \leq \overrightarrow{q} < \overrightarrow{Q}, \overrightarrow{0} \leq \overrightarrow{d} < \overrightarrow{D_{op}} \right\}$$

Pour pouvoir désigner l'ensemble des points consommés après le changement de pavage, nous devons d'abord décrire les principes de base de ce dernier. À l'image du changement de pavage par ajout de dimension, nous devons d'abord désigner le vecteur de pavage

que nous allons agrandir, puis choisir le facteur d'agrandissement. Là encore il faut que la borne d'itération du vecteur de pavage soit un multiple du facteur d'agrandissement. La différence vient du macro-ajustage dans lequel nous ne rajoutons pas de dimensions, mais dans lequel nous choisissons une dimension que nous décidons d'agrandir pour mettre les nouveaux éléments issus de l'agrandissement du macro-motif. On désigne par i l'indice du vecteur de pavage, par k l'indice du vecteur d'ajustage et par $\mathcal{L}$ la nouvelle borne d'itération du vecteur de macro-ajustage.

L'application du changement de pavage sur une tâche ne modifie donc que deux vecteurs de pavages et d'ajustages ainsi que les bornes d'itérations qui leurs sont associées. L'ensemble des itérations sur ces deux vecteurs est combiné avec l'emsemble des itérations sur les autres vecteurs pour produire l'ensemble des points consommés. Or si on modifie ces deux vecteurs, il suffit de vérifier que l'ensemble des valeurs qu'ils produisent avant et après la modification soient égales pour prouver que l'ensemble des points consommés reste le même. On peut donc se contenter de tester l'égalité des ensembles suivant :

$$U_{av} = \left\{ \overrightarrow{u} | \overrightarrow{u} \equiv \left(\overrightarrow{P_{op,i}} \times x_1 + \overrightarrow{\mathcal{M}F_{op,k}} \times y_1 \right) \mod \overrightarrow{M_{op}}, 0 \leq x_1 < Q_i, 0 \leq y_1 < \mathcal{M}D_{op,k} \right\}$$
$$U_{ap} = \left\{ \overrightarrow{u} | \overrightarrow{u} \equiv \left(n \times \overrightarrow{P_{op,i}} \times x_2 + \overrightarrow{\mathcal{M}F_{op,k}} \times y_2 \right) \mod \overrightarrow{M_{op}}, 0 \leq x_2 < \frac{Q_i}{n}, 0 \leq y_2 < \mathcal{L} \right\}$$

On peut tester l'égalité de ces deux ensembles à l'aide de l'équation :

$$\exists \mathcal{L} \in \mathbb{N}, \forall x \in \mathbb{N}, 0 \leq x < Q_i, \forall y_1 \in \mathbb{N}, 0 \leq y_1 < \mathcal{M}D_{op,k}, \exists y_2, 0 \leq y_2 < \mathcal{L}$$
$$\overrightarrow{P_{op,i}} \times x + \overrightarrow{\mathcal{M}F_{op,k}} \times y_1 \equiv n \times \overrightarrow{P_{op,i}} \times \lfloor \frac{x}{n} \rfloor + \overrightarrow{\mathcal{M}F_{op,k}} \times y_2 \mod \overrightarrow{M_{op}}$$

Cependant cette égalité n'est réellement équivalente aux précédentes que si l'ensemble des y_2 qu'elle définit, constitue l'ensemble des valeurs entières entre 0 et $\mathcal{L}$. En effet, l'intervalle $[0, \mathcal{L}[$ est en fait notre nouvel espace d'itération pour le vecteur d'ajutage k après le changement de pavage.

Nous nous contenterons de prouver cette égalité dans le cas où les tâches consomment les points à des itérations de pavage correspondantes. En effet, le changement de pavage consiste à grouper des macro-motifs. Ainsi avec un facteur d'agrandissement de n, on groupe n macro-motifs en un seul. Donc les n itérations de pavage servant à construire ces n macro-motifs sont regroupées en une seule itération. On en déduit qu'il existe une surjection f entre les itération de pavage avant et après le changement de pavage, f est definie de la manière suivante : en posant $E = \{x \in \mathbb{N} | 0 \leq x < Q_i\}$ et $F = \left\{ x \in \mathbb{N} | 0 \leq x < \frac{Q_i}{n} \right\}$, on a :

$$f : E \to F$$
$$x \to \lfloor \frac{x}{n} \rfloor$$

On peut écrire alors notre égalité sous la forme :

$$\overrightarrow{P_{op,i}} \left(x - n \times \lfloor \frac{x}{n} \rfloor \right) + \left(\overrightarrow{\mathcal{M}F_{op,k}} \times (y_1 - y_2) \right) \equiv \overrightarrow{0} \mod \overrightarrow{M_{op}}$$

On pose $x = tn + m$ avec $t \in \mathbb{N}$ et $m \in \mathbb{N}, 0 \leq m < n$ ce qui donne :

$$x - n \times \lfloor \frac{x}{n} \rfloor = tn + m - n \times \lfloor \frac{tn + m}{n} \rfloor = tn + m - nt = m$$

Finalement, on obtient :

$$\exists \mathcal{L} \in \mathbb{N}, \forall m \in \mathbb{N}, 0 \leq m < n, \forall y_1 \in \mathbb{N}, 0 \leq y_1 < \mathcal{M}D_{op,k}, \exists y_2 \in \mathbb{N}, 0 \leq y_2 < \mathcal{L}$$
$$\overrightarrow{P_{op,i}} \times m + \overrightarrow{\mathcal{M}F_{op,k}} \times (y_1 - y_2) \equiv \overrightarrow{0} \mod \overrightarrow{M_{op}}$$

On constate, à la vue de cette équation, que la condition d'égalité des ensembles consommés et produits avant et après le changement de pavage, est très ressemblante à celle de la présence d'un recalcul dans une tâche hiérarchique. En effet, si on décide d'utiliser pour le changement de pavage les dimensions de pavage et d'ajustage provoquant le recalcul, alors on obtient des équations quasi similaires. On peut donc se poser deux questions :

1. les ensembles consommés avant et après le changement de pavage par une tâche hiérarchique sont-ils bien les mêmes lorsque cette tâche comporte un recalcul par recouvrement ?
2. comment respecter les conditions de validité du changement de pavage du coté résultat puisque les propriétés du recouvrement s'appliquent uniquement à la partie opérande ?

3.4.5.3 Validité lorsque la tâche est hiérarchique

Supposons que nous venons d'appliquer un changement de pavage par agrandissement linéaire sur les vecteurs de pavages et d'ajustages qui provoquaient un recouvrement dans une tâche hiérarchique. Puisque la tâche est hiérarchique nous savons que :

$$\exists \Delta, -\mathcal{M}D_{op,k} < \Delta < \mathcal{M}D_{op,k}, \overrightarrow{P_{op,i}} - \overrightarrow{\mathcal{M}F_{op,k}} \times \Delta \equiv 0 \mod \overrightarrow{M_{op}}$$

On peut facilement modifier cette équation en écrivant (avec m et y_1 dans $\mathbb{N}$) :

$$\overrightarrow{P_{op,i}} \times m + \overrightarrow{\mathcal{M}F_{op,k}} \times (-m\Delta) \equiv \overrightarrow{0} \mod \overrightarrow{M_{op}}$$
$$\overrightarrow{P_{op,i}} \times m + \overrightarrow{\mathcal{M}F_{op,k}} \times \big(y_1 - (y_1 + m\Delta)\big) \equiv \overrightarrow{0} \mod \overrightarrow{M_{op}}$$

On chosit donc y_2 tel que :

$$\forall m, 0 \leq m < n, \forall y_1, 0 \leq y_1 < \mathcal{M}D_{op,k}, y_2 = y_1 + m\Delta$$

On a alors y_2 qui vérifie que :

$$\exists \mathcal{L} \in \mathbb{N}, \forall m \in \mathbb{N}, 0 \leq m < n, \forall y_1 \in \mathbb{N}, 0 \leq y_1 < \mathcal{M}D_{op,k}, \exists y_2 \in \mathbb{N}, 0 \leq y_2 < \mathcal{L}$$
$$\overrightarrow{P_{op,i}} \times m + \overrightarrow{\mathcal{M}F_{op,k}} \times (y_1 - y_2) \equiv \overrightarrow{0} \mod \overrightarrow{M_{op}}$$

La consommation des données avant et après le changement de pavage n'a donc pas changée. Il nous reste cependant à vérifier que l'ensemble des y_2 tel que nous les avons défini constitue bien l'ensemble des valeurs entières entre 0 et $\mathcal{L}$. Or nous avons défini y_2 comme étant égal à $y_1 + m\Delta$ comme Δ peut être positif ou négatif puisque sa valeur est comprise entre $-\mathcal{M}D_{op,k}$ et $\mathcal{M}D_{op,k}$ nous allons distinguer deux cas.

Cas n° 1 : $\Delta > 0$: Nous savons que $\forall m, 0 \leq m < n, \forall y_1, 0 \leq y_1 < \mathcal{MD}_{op,k}, \exists y_2, y_2 = y_1 + m\Delta$, il est facile de prouver par récurrence que y_2 prend toutes les valeurs entières dans l'intervalle $[0, \mathcal{MD}_{op,k} + (n-1)\Delta[$.

- Pour $n = 1$, on a $m = 0$ donc $\forall y_1, 0 \leq y_1 < \mathcal{MD}_{op,k}, y_2 = y_1$ donc y_2 prend toutes les valeurs entières entre 0 et $\mathcal{MD}_{op,k}$.
- Pour $n = i$ tel que pour $n = i - 1$ la condition de récurrence soit respectée.
  - Pour $0 \leq m < i - 1$, on se retrouve dans le cas $n = i - 1$, on sait alors que y_2 prend toutes les valeurs entières dans l'intervalle $[0, \mathcal{MD}_{op,k} + (i-2)\Delta[$.
  - Pour $m = i - 1$, on a $y2 = y1 + (i-1)\Delta$ et donc y_2 prend toutes les valeurs entières dans l'intervalle $[(i-1)\Delta, \mathcal{MD}_{op,k} + (i-1)\Delta[$.
  - Or comme $0 \leq \Delta < \mathcal{MD}_{op,k}$, on a $(i-1)\Delta < (i-2)\Delta + \mathcal{MD}_{op,k}$, donc l'ensemble des valeurs prises par y_2, pour les valeurs de m inférieures à $i - 1$, n'est pas disjoint de celle prise lorsque $m = i - 1$. La condition de récurrence est donc valable pour $m = i$.
- On a donc réussi à prouver que pour $\forall n \in \mathbb{N}$, on a y_2 qui prend toutes les valeurs entières dans l'intervalle $[0, \mathcal{MD}_{op,k} + (n-1)\Delta[$.

Les points consommés avant et après le changement de pavage sont donc équivalents lorsque $\Delta > 0$. On note également qu'il faut prendre $\mathcal{L} = \mathcal{MD}_{op,k} + (n-1)\Delta$.

Cas n° 2 : $\Delta < 0$: On démontre par un raisonnement similaire que y_2 occupe toutes les valeurs dans l'intervalle $[-(n-1)|\Delta|, \mathcal{MD}_{op,k}[$. Or y_2 doit être supérieur à 0, il nous faut donc effectuer un décalage de l'espace d'itération. On propose alors de décaler notre intervalle de $(n-1)|\Delta|$ et on obtient donc un intervalle valide : $[0, \mathcal{MD}_{op,k} + (n-1)|\Delta|[$. Comme le montre la section 2.4.3.1 page 57, il faut introduire un *shift* de valeur $(-(n-1)|\Delta| \times \overrightarrow{\mathcal{MF}_{op,k}})$ pour « contrecarrer » les effets de ce décalage.

$$U_{ap} = \left\{ \begin{array}{l} \overrightarrow{u} | \overrightarrow{u} \equiv \left((-(n-1)|\Delta| \times \overrightarrow{\mathcal{MF}_{op,k}}) + n \times \overrightarrow{P_{op,i}} \times x_2 + \overrightarrow{\mathcal{MF}_{op,k}} \times y_2 \right) \mod \overrightarrow{M_{op}}, \\ 0 \leq x_2 < \frac{Q_i}{n}, 0 \leq y_2 < \mathcal{MD}_{op,k} + (n-1)|\Delta| \end{array} \right\}$$

3.4.5.4 Conclusion sur la validité

Nous avons démontré qu'il est possible d'avoir une égalité en U_{av} et U_{ap}. Cependant dans le cas où $\Delta < 0$, l'utilisation du changement de pavage introduit une difficulté qui n'est pas présente dans les équations précédentes.

En effet, le changement de pavage permet d'agrandir un macro-motif pour qu'il puisse contenir au moins un autre macro-motif. On peut donc distinguer les élements du premier macro-motif de ceux du deuxième et ceci même si certains éléments sont communs au deux. Or lorsque $\Delta < 0$, $y_2 = y_1 - m|\Delta|$, les origines des macro-motifs sont alors atteintes[5] pour $y_2 = 0$ et pour $y_2 = -|\Delta|$. Comme $-|\Delta| < 0$, les élements du premier macro-motif ne seront plus rangés en premières positions dans le nouveau macro-motif donc si on ne modifie pas les sous-tâches et qu'on se contente d'agrandir les dimensions de macro-ajustage, alors la deuxième sous-tâche va ranger elle aussi ses résultats dans le mauvais ordre. Il nous faut donc rétablir l'ordre des éléments dans un macro-motif agrandi.

[5] On ne tient pas compte ici du décalage.

3.4.6 Changement de pavage par agrandissement linéaire : partie résultat

Nous avons vu dans la section précédente comment agrandir un macro-motif dans le cadre du changement de pavage par agrandissement linéaire. Ce type d'agrandissement évite le recalcul puisque contrairement à l'agrandissement par ajout de dimensions, les motifs originels (ceux existants avant la fusion) ne sont présents qu'une seule fois.

Une fois ce macro-motif agrandi, il est passé à la première sous-tâche de la hiérarchie qui va consommer un à un les motifs originels dont il est constitué. Puis pour chacun de ces motifs, cette première sous-tâche va produire un motif résultat. La deuxième sous-tâche va alors consommer par bloc ces motifs résultats pour produire autant de blocs résultats. C'est ici au niveau de la consommation de la deuxième sous-tâche que nous allons gérer le problème d'ordre des éléments dans les macro-motifs.

Pour simplifier les explications ci-dessous, on considére que $n = 2$, le macro-motif issu de l'agrandissement est donc composé de deux macro-motifs originaux. On désigne par le terme de premier macro-motif les éléments qui se trouvaient déja dans le macro-motif avant l'agrandissement, et par le terme de deuxième macro-motif les éléments qui constituaient le macro-motif qui a été ajouté.

La solution que nous proposons est relativement simple. On se base au niveau de la première sous-tâche, nous rappelons que celle-ci est construite de la manière suivante : son pavage est constitué du macro-ajustage de la tâche supérieure et son ajustage est constitué de l'ajustage spatial de tâche supérieure. Ainsi la dimension de macro-ajustage associée au vecteur $\overrightarrow{\mathcal{M}F_{op,k}}$ devient une dimension de pavage et elle comporte donc un vecteur de pavage opérande et un vecteur de pavage résultat. On nomme ces deux vecteurs respectivement $\overrightarrow{\mathcal{S}P_{op1,k}}$ et $\overrightarrow{\mathcal{S}P_{res1,k}}$.En outre nous considérons que la projection de la partie opérande est égale à la matrice identité comme lorsque la hiérarchie est issue d'une fusion. Si cette condition n'est pas respectée, il est toujours possible de se ramener à un cas semblable comme nous l'avons montré dans la section 2.4.3.3 page 61.

Nous partons de l'origine du premier motif originel du premier macro-motif. Lorsque $\Delta > 0$ cette origine est atteinte pour une valeur d'itération égale à 0 sur $\overrightarrow{\mathcal{S}P_{op1,k}}$. Lorsque $\Delta < 0$, elle est atteinte pour une itération égale à $(n-1)|\Delta|$. On calcule alors les coordonnées de l'origine du motif résultat issu de la consommation de ce premier motif originel par la première sous-tâche. Ce calcul est donc effectué pour une itération de pavage sur $\overrightarrow{\mathcal{S}P_{res1,k}}$ égale soit à 0 soit à $(n-1)|\Delta|$ et ceci en fonction du signe de Δ. Si il y a d'autres vecteurs de pavage on laisse leur nombre d'itération à 0 par mesure de commodité. On obtient alors soit $S = \overrightarrow{\mathcal{S}P_{res1,k}} \times 0 = \overrightarrow{0}$, soit $S = \overrightarrow{\mathcal{S}P_{res1,k}} \times (n-1)|\Delta|$. Si nous rajoutons S au *shift* de la partie opérande de la deuxième sous-tâche et nous apportons aucune autre modification à cette dernière, alors nous allons produire exactement les mêmes résultats qu'avant le changement de pavage. Que Δ soit positif ou négatif, on traite ainsi les éléments calculés à partir du premier macro-motif. Il nous faut donc maintenant faire de même pour les éléments issus du deuxième macro-motif.

On propose simplement d'ajouter une nouvelle dimension de pavage à notre deuxième sous-tâche. Cette dimension a pour but de permettre de passer des motifs résultats issus des calculs sur le premier macro-motif aux motifs issus des calculs sur le deuxième macro-motif. Pour obtenir ce nouveau vecteur, on calcule les coordoonées de l'origine du premier motif issu des calculs sur le deuxième macro-motif. Par la même méthode que celle utilisée pour calculer S, on obtient $\overrightarrow{\mathcal{S}P_{res1,k}} \times \Delta$ si $\Delta > 0$, et $\overrightarrow{\mathcal{S}P_{res1,k}} \times (n-2)|\Delta|$ si $\Delta < 0$. Puis à l'aide de

ces coordonnées et de celles calculées pour S, on déduit la valeur de notre nouveau vecteur de pavage.

- Pour $\Delta > 0$, $\overrightarrow{\mathcal{S}P_{res1,k}} \times \Delta - \overrightarrow{0} = \overrightarrow{\mathcal{S}P_{res1,k}} \times \Delta$.
- Pour $\Delta < 0$, $\overrightarrow{\mathcal{S}P_{res1,k}} \times (n-2)|\Delta| - \overrightarrow{\mathcal{S}P_{res1,k}} \times (n-1)|\Delta| = -\overrightarrow{\mathcal{S}P_{res1,k}} \times |\Delta|$.

Conclusion le nouveau vecteur de pavage vaut $\overrightarrow{\mathcal{S}P_{res1,k}} \times \Delta$ quel que soit Δ. Si le changement de pavage agrandit le macro-motif pour qu'il contienne plus de 2 macro-motifs originaux, c'est-à-dire si $n > 2$, il est facile de prouver que notre vecteur de pavage reste valide. La borne d'itération associée à ce vecteur est bien sûr égale à n. Nous venons de calculer un vecteur de pavage opérande, il nous faut également ajouter un vecteur de pavage résultat.

Pour nous simplifier la vie et pour respecter le fait que la segmentation résultat est égale à la matrice identité après une fusion, nous ajoutons une dimension spatiale au tableau de sortie de la sous-tâche gauche. Ainsi notre vecteur de pavage résultat est de la forme $\begin{pmatrix} 1 \\ 0 \\ \dots \end{pmatrix}$ comme dans le cadre du changement de pavage par ajout de dimensions. Cette nouvelle dimension spatiale est utilisée comme une rangée de « stockage » des résultats issus de chaque macro-motif.

Au niveau de la partie résultat de la tâche supérieur, il nous suffit de faire passer la dimension de pavage correspondant à $\overrightarrow{P_{op,i}}$ dans l'ajustage comme pour un changement de pavage par ajout de dimensions.

3.4.7 Changement de pavage par agrandissement linéaire

3.4.7.1 Utilisation

Dans la section précédente, nous avons détaillé le raisonnement qui nous a permis d'établir les principes du changement de pavage par agrandissement linéaire. Nous présentons ici les différentes étapes nécessaires à l'application d'un tel changement de pavage. Notre point de départ est la représentation ODT d'une tâche hiérarchique. Le premier paramètre qui doit nous être donné est le facteur d'agrandissement n.

Puis, il faut choisir la dimension de pavage à agrandir ($\overrightarrow{P_{op,i}}$), mais il est possible d'avoir un recouvrement pour plusieurs dimensions de pavage. Par exemple dans un tableau bidimensionnel, il est facile d'imaginer une tâche dont les macro-motifs se recouvrent à la fois pour un vecteur de pavage horizontal et pour un vecteur de pavage vertical. Or le choix entre ces deux vecteurs peut ne pas être uniquement lié à la réduction du nombre de recalculs. En effet si un des vecteurs représente une dimension temporelle et l'autre une dimension spatiale, le choix de l'agrandissement de l'un ou de l'autre aura des conséquences très différentes. On peut donc soit laisser l'utilisateur nous le donner soit en choisir un par nous même.

De même, il peut également y avoir plusieurs choix possibles pour le vecteur de macro-ajustage opérande ($\overrightarrow{\mathcal{M}F_{op,k}}$). On peut prendre comme exemple une tâche hiérarchique dont la partie opérande de la tâche supérieure est de la forme suivante[6] :

$$(\infty)_M . (0)_S . |4 \quad | \quad 4 \quad 2 \quad | \quad 1| . \left(\begin{array}{|c|}\hline \infty \\ \hline 2 \\ 3 \\ \hline 2 \\ \hline \end{array}\right)_{\mathbf{G}} .$$

[6] Le pavage, le macro-ajustage et l'ajustage spatiale ont été séparés pour plus de clarté.

Mais cette fois-ci, le choix du vecteur est une simple affaire de performance du changement de pavage. Nous pouvons donc le calculer nous même à chaque fois.

Voici un algorithme simpliste permettant de trouver i, k et Δ :

```
Données : P_op la matrice de pavage opérande de la tâche supérieur.
          Q le vecteur des bornes de pavage
          MF_op les vecteurs de macro-ajustage de la matrice d'ajustage F_op
          MD_op le vecteur des bornes de macro-ajustage
          M_op le vecteur contenant taille du tableau opérande

Retour :  R l'ensemble des valeurs pouvant être prises par :
          - i l'index du vecteur de pavage dans P_op
          - k l'index du vecteur de macro-ajustage dans MF_op
          - Δ

Pour i Allant_de 0 A LongueurDe(Q) Faire
  Pour k Allant_de 0 A LongueurDe(MD) Faire
    Pour Δ Allant_de −MD[i] A MD[i] Faire
      Si P_op[i] = F_op,k × Δ mod M_op Alors
        R+ = {i, k, Δ}
      FinSi
    FinPour
  FinPour
FinPour
Retourne R
```

Algorithme 3.1: Calcul de i, de k et de Δ

Une fois que nous avons déterminé i, k et Δ , il faut calculer la valeur de $\mathcal{L}$ qui rappelons le vaut $\mathcal{MD}_{op,k}+(n-1)|\Delta|$ et la valeur de S qui vaut $\overrightarrow{0}$ si $\Delta > 0$ et $\overrightarrow{\mathcal{S}P_{res1,k}}\times(n-1)|\Delta|$ si $\Delta < 0$. Il faut également déterminer le nouveau vecteur de pavage de la sous-tâche gauche qui vaut $\overrightarrow{\mathcal{S}P_{res1,k}}\times\Delta$. Enfin il faut modifier notre tâche hiérarchique en fonction des résultats que nous venons d'obtenir. Les modifications à effectuer sont détaillées dans la section suivante.

3.4.7.2 Représentations ODT

On décrit ci-dessous la représentation ODT des différentes parties d'une tâche hiérarchique avant et après l'application du changement de pavage par agrandissement linéaire.

– Représentation de la tâche supérieure avant : on a volontairement introduit des séparations entre les parties de pavage, d'ajustage opérande et d'ajustage résultat afin de faciliter la compréhension des changements après l'application du changement de pavage.

$$(M_{op})_{\mathbf{M}}.(S_{op})_{\mathbf{S}}.\left|P_{op} \;\middle|\; \mathcal{MF}_{op} \;\; F_{op} \;\middle|\; 0 \;\; 0\right|.\left(\begin{array}{c} Q \\ \hline \mathcal{MD}_{op} \\ D_{op} \\ \hline \mathcal{MD}_{res} \\ D_{res} \end{array}\right)_{\mathbf{G}}.\left|\begin{array}{c|c|cc} P_{res} & 0 \;\; 0 & \mathcal{MF}_{res} & F_{res} \end{array}\right|.$$

– Représentation de la tâche supérieure après : on constate l'agrandissement des vecteurs de pavage, l'augmentation du nombre d'itération sur le k^{e} vecteur de macro-

ajustage opérande et le glissement du i^{e} vecteur de pavage résultat dans le macro-ajustage et l'apparition d'une dimension de taille n qui en découle.

$$\left(M_{op}\right)_{\mathbf{M}}.\left(S_{op}\right)_{\mathbf{S}}.\left|\ldots \quad n\times P_{op,i} \quad \ldots \mid \ldots \quad \mathcal{M}F_{op,k} \quad \ldots \quad F_{op} \quad \middle| \quad 0 \quad 0 \quad 0\right|$$
$$.\begin{pmatrix}\vdots\\ Q_i/n\\ \vdots\\ \hline \vdots\\ \mathcal{L}\\ \vdots\\ D_{op}\\ \hline n\\ \mathcal{M}D_{res}\\ D_{res}\end{pmatrix}_{\mathbf{G}} .\overline{\underline{\ldots. \quad n\times P_{res,i} \quad \ldots \quad \middle| \quad 0 \quad 0 \quad 0 \quad 0 \quad \middle| \quad P_{res,i} \quad \mathcal{M}F_{res} \quad F_{res}}}\ .$$

– Représentation de la première sous-tâche avant :

$$\begin{pmatrix}\mathcal{M}D_{op}\\ D_{op}\end{pmatrix}_{\mathbf{M}}.\begin{pmatrix}0\\0\end{pmatrix}_{\mathbf{S}}.\begin{vmatrix}1&0&0\\0&1&0\end{vmatrix}.\begin{pmatrix}\mathcal{M}D_{op}\\ D_{op}\\ \mathcal{S}D_{res1}\end{pmatrix}_{\mathbf{G}}.\overline{\underline{\mathcal{S}P_{res1} \quad 0 \quad \mathcal{S}F_{res1}}}\ .$$

– Représentation de la première sous-tâche après : il suffit juste d'agrandir la k^e dimension de pavage correspondant à la k^e dimension de macro-ajustage opérande dans la tâche supérieure.

$$\begin{pmatrix}\vdots\\ \mathcal{L}\\ \vdots\\ D_{op}\end{pmatrix}_{\mathbf{M}}.\begin{pmatrix}0\\0\end{pmatrix}_{\mathbf{S}}.\begin{vmatrix}1&0&0\\0&1&0\end{vmatrix}.\begin{pmatrix}\vdots\\ \mathcal{L}\\ \vdots\\ D_{op}\\ \mathcal{S}D_{res1}\end{pmatrix}_{\mathbf{G}}.\overline{\underline{\mathcal{S}P_{res1} \quad 0 \quad \mathcal{S}F_{res1}}}\ .$$

– Représentation de la deuxième sous-tâche avant :

$$\left(M_{op2}\right)_{\mathbf{M}}.\left(S_{op2}\right)_{\mathbf{S}}.\left|\mathcal{S}P_{op2} \quad 0 \quad \mathcal{S}F_{op2}\right|.\begin{pmatrix}\mathcal{M}D_{res}\\ \mathcal{S}D_{op2}\\ D_{res}\end{pmatrix}_{\mathbf{G}}.\overline{\underline{\begin{matrix}1&0&0\\0&0&1\end{matrix}}}\ .$$

– Représentation de la deuxième sous-tâche après : coté opérande, on voit l'apparition du nouveau vecteur de pavage et l'ajout de S au *shift*. Côté résultat, on constate que l'ajout du nouveau vecteur de pavage se combine avec l'ajout d'une dimension spatiale de taille n dans le tableau résultat.

$$\left(M_{op2}\right)_{\mathbf{M}}.\left(S_{op2}+S\right)_{\mathbf{S}}.\left|\mathcal{S}P_{res1,k}\times\Delta \quad \mathcal{S}P_{op2} \quad 0 \quad \mathcal{S}F_{op2}\right|.\begin{pmatrix}n\\ \mathcal{M}D_{res}\\ \mathcal{S}D_{op2}\\ D_{res}\end{pmatrix}_{\mathbf{G}}.\overline{\underline{\begin{matrix}1&0&0&0\\0&1&0&0\\0&0&0&1\end{matrix}}}\ .$$

3.4.7.3 Évaluation de la réduction des recalculs

L'équation 3.2 page 87, celle du recouvrement, nous indique qu'il faut $|\Delta|$ itérations d'ajustage sur $\overrightarrow{\mathcal{MF}_{op,k}}$ pour atteindre une itération de pavage sur $\overrightarrow{P_{op,i}}$. Or il y a en tout $\mathcal{MD}_{op,k}$ itérations d'ajustage possibles sur cette dimension pour constituer un macro-motif. Le taux de recouvrement est donc égal à $\frac{\mathcal{MD}_{op,k}}{|\Delta|}$. Pour évaluer ce taux de recouvrement après un changement de pavage, il suffit d'appliquer le même raisonnement en prenant $\mathcal{L}$ comme nombre total d'itérations d'ajustage et en multipliant Δ par n comme cela a été le cas pour $\overrightarrow{P_{op,i}}$. On obtient alors un taux de recouvrement égal à :

$$\frac{\mathcal{MD}_{op,k} + (n-1)|\Delta|}{n|\Delta|} \tag{3.3}$$

À chaque recouvrement correspond une origine de macro-motif et donc une exécution de la première sous-tâche, ce taux de recouvrement fait donc également office de taux de recalculs. Idéalement ce taux doit tendre vers 1, pour qu'il n'y ait qu'une seule exécution de la première sous-tâche sur les données d'un même macro-motif. Attention toutefois, ce taux n'est pas valable pour tous les macro-motifs d'une tâche ARRAY-OL. En effet, les premiers et les derniers macro-motifs ne souffrent pas du problème de recouvrement puisqu'il n'y a pas d'autres macro-motifs avant ou après eux (*cf* figures 3.14 page 81 et 3.15 page 83).

3.4.8 Conclusion

En étudiant les conséquences de la fusion, nous avons vu que dans certains cas des recalculs pouvaient être introduits. Afin de diminuer leurs effets Julien Soula avait utilisé une transformation appelée changement de pavage. Mais son approche était limitée à des recalculs issus de fusions, nous avons alors proposé une nouvelle version du changement de pavage pouvant être utilisé dans tous les cas.

3.5 Applications particulières du changement de pavage

3.5.1 L'aplatissement

Dans le cadre du changement de pavage, on note n le facteur d'agrandissement du macro-motif. Il doit être choisi de manière à ce qu'il soit un diviseur de Q_i pour qu'on puisse grouper un nombre entier de macro-motif en un seul. Si $n = 1$, le changement de pavage n'a pas lieu d'être puisqu'il n'y a pas de modifications à effectuer. Mais que se passe-t-il si $n = Q_i$?

En fait, on groupe au niveau de la tâche supérieure tous les macro-motifs présents sur la i^e dimension de pavage en un seul « gros » macro-motif. Or s'il n'y a qu'une seule dimension de pavage, on se retrouve alors à consommer tout le taleau opérande en une seule fois et donc à produire tout le taleau résultat. La hiérarchie n'est donc plus utile. On peut alors se servir des résultats démontrés à la section 2.4.3.3 page 61 pour supprimer cette hiérarchie et remonter toutes les sous-tâches d'un niveau. Si la tâche comporte plusieurs dimensions de pavage il suffit d'appliquer le changement de pavage sur chacune de ces dimensions en prenant à chaque fois n égal à leur borne d'itération.

L'utilisation du changement de pavage pour supprimer une hiérarchie est appelée aplatissement *(collapse)* et constitue la troisième des transformations à notre disposition pour

modifier une application ARRAY-OL. L'aplatissement est particulièrement utile pour lutter contre la création de hiérarchies abyssales créées par des fusions successives. Les figures 3.17, 3.18, 3.19 et 3.20 montrent deux fusions successives qui créent une hiérarchie à deux niveaux ; on utilise alors l'aplatissement pour se ramener à une hiérarchie à un seul niveau. Il est important de remarquer que l'aplatissement ne diminue pas les effets de la fusion : la taille des tableaux de la sous-tâche est minimum pour pouvoir passer directement de $T1$ à $T3$. L'utilisation conjointe de la fusion et l'aplatissement permet en réalité de fusionner une suite de tâches au lieu de deux.

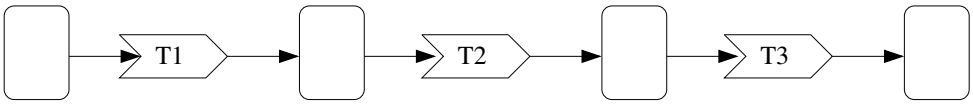


FIG. 3.17: La tâche originale

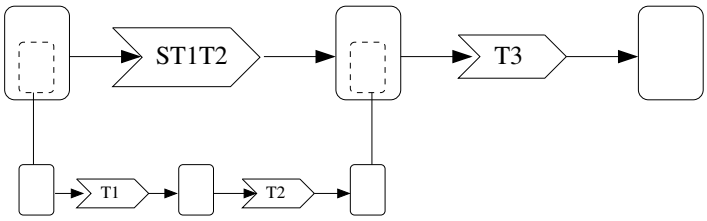


FIG. 3.18: Après la première fusion

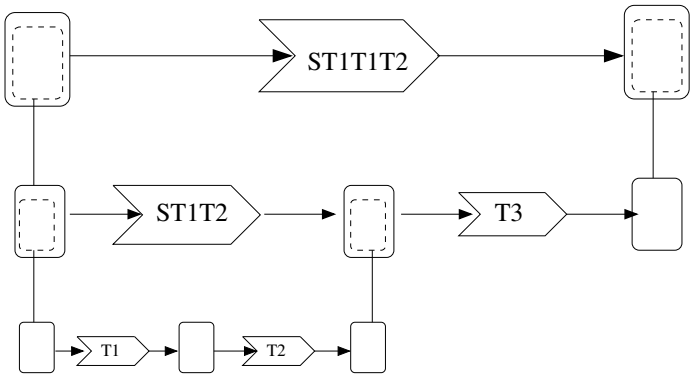


FIG. 3.19: Après la deuxième fusion

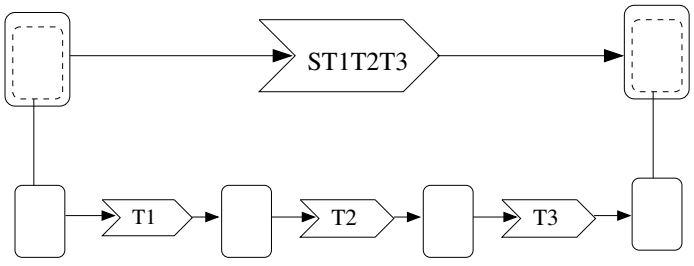


FIG. 3.20: Résultat final

3.5.2 La mise à niveau

Il est donc possible d'utiliser la fusion et l'aplatissement pour réaliser des fusions multiples qui ne comportent au final qu'un seul niveau de hiérarchie. On nomme mise à niveau *(one-level)* l'utilisation conjointe de la fusion et de l'aplatissement, ce qui constitue la quatrième transformation ; cette transformation prend en paramètre n tâches et elle transforme ces tâches en une tâche hiérarchique simple. On obtient alors une application qui respecte la forme suivante :

1. le premier niveau comporte une tâche qui consomme les tableaux opérandes de l'application originelle et qui produit ses tableaux résultats ;
2. le deuxième niveau de la hiérarchie est constitué des tâches originelles.

On peut noter en outre que l'utilisation de la mise à niveau limite la présence des tableaux infinis au premier niveau. En effet, lors d'une fusion, les tableaux de la sous-tâche sont des motifs de la tâche supérieure, ils ne peuvent donc pas être de taille infinie. La mise à niveau qui est une suite de fusions interdit donc la présence de tableaux infinis dans la sous-tâche. Au final, les tableaux opérandes ou résultats d'une application sont les seuls à pouvoir contenir des dimensions infinies et ils ne sont présents qu'au premier niveau de hiérarchie.

3.6 Le tiling

Nous venons de voir la transformation dite d'aplatissement qui permet de supprimer une hiérarchie. Nous présentons dans la section suivante une transformation capable de faire l'opération inverse, à savoir créer une dimension.

Principe du tiling : Contrairement à la fusion qui nécessite deux tâches et pour laquelle la création de hiérarchie n'est qu'une conséquence indirecte, le tiling est une transformation qui a pour unique but de créer une hiérarchie à partir d'une seule tâche. Son principe est extrêment simple, il consiste à se servir des motifs de la tâche supérieure comme des tableaux d'entrée et de sortie de la sous-tâche. Cette sous-tâche consomme et produit directement ce motif avec des matrices d'ajustage égales à la matrice identité. D'ailleurs on peut noter que la sous-tâche ne comporte pas de pavage puisqu'elle consomme directement le motif. Le fonctionnement du tiling est illustré par la figure 3.21 .

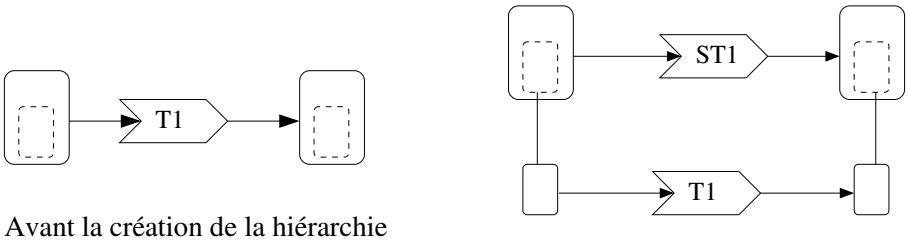
Le tiling n'est pas une transformation visant à l'optimisation de l'application, il est en fait un outil supplémentaire pour transformer notre application selon nos besoins. On peut également noter qu'il fonctionne de la même façon que la transformation de boucles appelée changement de pavage(*cf* section 2.3.2.2 page 47).

Équations ODT : La représentation ODT du tiling est immédiate :

– Représentation de la tâche supérieure :

$$\left(M_{op}\right)_{\mathbf{M}} . \left(S_{op}\right)_{\mathbf{S}} . \left|P_{op} \quad F_{op} \quad 0\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} P_{res} & 0 & F_{res} \end{matrix}} \quad .$$

– Après application du tiling, la représentation de la tâche supérieure ne change pas car il n'y a pas de modifications des dépendances de données : les tableaux et les motifs consommés et produits par la tâche sont toujours les mêmes.



Avant la création de la hiérarchie

Après la création de la hiérarchie

FIG. 3.21: Application du tiling

– Au niveau de la sous-tâche, le tableau d'entrée est le motif opérande de la tâche supérieure, mais il est aussi le motif opérande de la sous-tâche ; on a donc un tableau de taille D_{op} qui est consommé par une matrice d'ajustage égale à la matrice identité. Un raisonnement symétrique s'applique pour construire la partie résultat.

$$\left(D_{op}\right)_{\mathbf{M}} . \left(0\right)_{\mathbf{S}} . \left|1 \quad 0\right| . \begin{pmatrix} D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{0 \quad 1} . \left(D_{res}\right)_{\star} .$$

3.7 Conclusion

Nous avons présenté dans ce chapitre cinq transformations : la fusion, le changement de pavage, l'aplatissement, la mise à niveau et le tiling ; nous avons étudié leurs principes et leurs utilisations. Ils constituent une « boîte à outils » capable d'effectuer de simples modifications ou des optimisations plus complexes sur des applications ARRAY-OL. Nous verrons d'ailleurs leur intérêt dans le cadre de l'exécution de ces applications.

Si nous reprenons notre comparaison avec les transformations de boucles (*cf* section 2.3.1 page 42), nous pouvons conclure que les transformations que nous proposons sont de plus haut niveau que les transformations de boucles usuelles car elles répondent plus directement aux problèmes. Ainsi pour optimiser la mémoire, il faut combiner plusieurs transformations de boucles élémentaires dans un ordre complexe alors que la mise à niveau suffit. Nos transformations sont donc plus proches d'une description à haut niveau des applications ARRAY-OL, alors que les transformations de boucles sont plus proches de leurs implémentations.

Enfin, ce chapitre contient plusieurs apports importants de cette thèse : à savoir une réécriture complète de la démonstration de la fusion, une implémentation de cette fusion, une nouvelle version du changement de pavage et l'apparition d'une nouvelle transformation le tiling.

Chapitre 4

Exécution d'ARRAY-OL

बंदर क्या जाने अदरक का स्वाद

Proverbe hindi.

4.1 Introduction

Dans le précédent chapitre, nous avons vu que pour améliorer l'exécution d'une application ARRAY-OL, il était préférable de l'optimiser en la modifiant au préalable. Nous avons alors proposé un ensemble d'outils pour permettre ces modifications. Le but de ce chapitre est de montrer l'impact de l'utilisation de ces outils dans le cadre d'une exécution sur différents modèles de calculs.

Dans la première section, nous allons étudier la projection d'ARRAY-OL sur des modèles de calculs « génériques », puis dans les deux sections suivantes nous analyserons une projection sur deux modèles de calculs concrets : SDF et KPN.

4.2 Projection d'ARRAY-OL sur un modèle de calcul

Il y a de nombreux modèles de calculs, mais ils reposent, pour la plupart, sur un des trois paradigmes suivants : séquentiel, SPMD et pipeline. Nous allons donc étudier la projection d'ARRAY-OL sur ces trois paradigmes.

4.2.1 Projection « naïve »

On peut facilement proposer une projection « naïve » d'ARRAY-OL sur les modèles de calculs suivant :

- Séquentiel : la projection est évidente ; au niveau du modèle global les tâches sont exécutées une par une, au niveau du modèle local les motifs sont consommés et produits un à un.
- SPMD : les itérations de pavage d'un modèle local étant indépendantes, il est possible de produire des motifs en parallèle. En revanche au niveau du modèle global, il faut qu'une tâche ait fini de produire son tableau de sortie pour pouvoir exécuter la tâche suivante, il n'y a donc pas de data-parallélisme possible.

– Pipeline : utiliser un pipeline au niveau d'un modèle local ne présente pas d'intérêt particulier, et pipeliner les tableaux au niveau du modèle global n'est pas possible, chaque tâche consommant et produisant un seul tableau.

Il semble donc à première vue que la projection la plus intéressante est une utilisation conjointe du séquentiel et du SPMD. Le modèle global est ainsi exécuté séquentiellement alors que le modèle local bénéficie de l'expression du data-parallélisme présent dans ARRAY-OL. Mais une exécution séquentielle du modèle global reste peu intéressante : une seule tâche étant exécutée à la fois. De plus l'utilisation de dimensions infinies dans les tableaux implique une exécution également infinie des tâches et donc provoque un blocage de l'exécution. En outre, nous utilisons les dimensions infinies pour représenter le temps, il est alors évident que ces dimensions sont en fait séquentialisées dans un cas concret. Il serait alors intéressant de proposer une séquentialisation des tableaux pour permettre la création d'un flux et donc de bénéficier d'une exécution de type pipeline. Cette méthode offrirait d'ailleurs l'avantage de rapprocher ARRAY-OL des modèles à flux de données utilisés pour représenter les applications du traitement du signal systématique (*cf* section 1.1.2 page 5).

4.2.2 Création d'un « flux »

La principale question que nous avons à nous poser sur la création du flux est le type des données qu'il va transporter. Devons nous créer un flux de motifs ou un flux de tableaux ?

4.2.2.1 Flux de motifs

Créer un flux de motifs entre les tâches est la première idée qui vient à l'esprit. Chaque tâche consomme alors ses motifs à partir de son flux entrant et relâche directement les motifs qu'elle produit sur son flux sortant.

Mais on rencontre très vite le problème suivant : les tableaux peuvent être produits et consommés de manières différentes, en d'autres termes les motifs de sortie d'une tâche ne sont pas nécessairement les motifs d'entrée de la tâche suivante. Le cas, dit du *corner turn*, où un tableau est produit ligne par ligne par une tâche et consommé colonne par colonne par la tâche suivante, est un exemple caractéristique de ce problème (*cf* figure 1.24 page 31). Pour surmonter cette difficulté, il nous faut donc grouper des motifs de sortie afin de former au moins un motif d'entrée de la tâche suivante. Ce n'est pas un travail facile, mais c'est réalisable grâce à la transformation appelée fusion comme nous l'avons vu dans la section 3.2 page 63. La fusion permet de produire directement le résultat d'une tâche à partir de la partie opérande de la tâche précédente en regroupant ces deux tâches en une seule et en créant une hiérarchie. On peut alors envisager de fusionner deux à deux les tâches jugées incompatibles, c'est-à-dire les tâches pour lesquelles les motifs de sortie de la première ne seraient pas les motifs d'entrée de la deuxième.

Mais une fois ce problème résolu, nous devons encore trouver un ordre pour la production et la consommation des données qui permette de pipeliner les tâches de manière efficace. Or la présence éventuelle de tableaux infinis n'est pas résolue par la simple fusion des tâches incompatibles et ils peuvent encore occasionner des blocages. Prenons l'exemple d'un tableau bidimensionnel de taille $\binom{4}{\infty}$: si il consomme des motifs de taille $\binom{2}{2}$ suivant la dimension infinie alors il ne consommera que la moitié de ce tableau et il ne produira que la moitié du tableau résultat. L'ordre de consommation et donc de production des motifs est un réel problème que les tableaux comportent ou non une dimension infinie.

On peut alors envisager une autre solution qui bien que très semblable à celle-ci offre la possibilité de résoudre les problèmes que nous avons rencontré, tout en améliorant la lisibilité de l'application.

4.2.2.2 Flux de tableaux

L'alternative que nous proposons ici est la construction d'un flux de tableaux par application sur la totalité de l'application de la transformation dite de mise à niveau. Cette transformation que nous avons décrite dans la section 3.5.2 page 97 est une succession de fusions et d'aplatissement, il en résulte la création d'une application à deux niveaux de hiérarchies :

1. le premier de ces niveaux comporte seulement une tâche qui consomme les tableaux opérandes de l'application originelle et qui produit ses tableaux résultats ;
2. le deuxième niveau de la hiérarchie est constitué des tâches originelles, mais on remarque que les tableaux opérandes et résultats de ce niveau sont en fait les motifs consommés et produits par le premier niveau. L'application est donc définie au second niveau de hiérarchie.

Il est alors très facile de créer un flux en pipelinant les motifs opérande de la tâche supérieure qui deviendront les tableaux opérandes de notre application. On parle alors de flux de tableaux. L'utilisation de la transformation de mise à niveau nous garantit d'avoir au premier niveau des motifs minimums et donc de produire « le plus rapidement possible » un motif résultat.

De plus cette utilisation des transformations nous garantit que seul des tableaux du premier niveau peuvent avoir une dimension infinie comme nous l'avons montré dans la section 3.5.2 page 97. Ceci nous apporte deux avantages :

- il n'y a pas de blocages dûs à l'exécution infinie d'une tâche. Le premier niveau de hiérarchie ne comporte qu'une seule tâche qui ne fait qu'« écouler » le flux.
- les tableaux infinis au premier niveau de hiérarchie ne sont jamais représentés en mémoire lors d'une implémentation. On se contente en fait de « nourrir » le flux directement avec des motifs issus de ces tableaux. On peut alors réaliser plus aisément une implémentation grâce à l'absence de tableaux infinis.

Mais nous sommes encore soumis aux problèmes d'ordres d'exécution. En effet, il est toujours possible au premier niveau d'avoir un tableau de taille $\binom{4}{\infty}$ consommé avec des motifs de taille $\binom{2}{2}$ et on rencontre donc le même problème qu'avec les flux de motifs. Pour résoudre ce problème, on utilise le changement de pavage pour agrandir le motif et englober la ou les dimensions « prioritaires » (dans notre exemple le motif $\binom{2}{2}$ devient un motif $\binom{4}{2}$). Mais cette solution diminue l'efficacité du pipeline en augmentant la taille des tableaux. On peut utiliser la transformation appelée tiling pour introduire un niveau de hiérarchie intermédiaire qui redécoupe le motif qu'on vient d'agrandir (ici le motif $\binom{4}{2}$ est découpé en deux motifs $\binom{2}{2}$). L'ordre d'exécution est donc facilement définissable par cette méthode. En outre il n'y a pas de problèmes au deuxième niveau puisque nous savons par construction que ce dernier fournit une production de résultats optimisés en fonction des dépendances de données et que tous les calculs effectués sont indispensables à l'obtention d'un résultat.

Nous avons réussi à projeter de l'ARRAY-OL sur un modèle pipeline, mais l'utilisation de la mise à niveau offre également une projection intéressante sur un modèle SPMD. En effet, la tâche présente au premier niveau peut, comme toute tâche ARRAY-OL, être exécutée de manière data-parallèle. On peut donc exécuter indépendamment tous les deuxièmes niveaux.

De plus les tâches du deuxième niveau peuvent être elles mêmes exécutées de manière data-parallèle.

4.2.2.3 Impact sur l'expressivité des applications

Nous avons transformé une application afin de pouvoir la projeter intelligemment sur un modèle de calcul pipeline ou SPMD. Nous avons changé la forme de cette application mais nous n'en avons pas changé le sens puisqu'elle produit toujours les mêmes résultats. On est cependant en droit de se demander si la modification de sa forme ne rend pas l'application moins compréhensible. La réponse est clairement non et ceci pour trois raisons :

1. le deuxième niveau de hiérarchie est une copie parfaite de l'application avant sa transformation : l'ordre des tâches est le même, les motifs manipulés sont identiques, seule la taille des tableaux dont sont extraits les motifs a changé ;
2. la mise en évidence d'un flux de tableaux entre les deux niveaux permet de clarifier l'exécution de l'application et rapproche ARRAY-OL des modèles à flux de données traditionnels ;
3. la spécification de l'ordre d'exécution au premier niveau apporte des informations qui ne sont pas présentes dans les langages de flux. Ainsi si l'on reprend l'exemple de l'application sur le passage d'un signal télévisé du format $4/3$ au format $16/9$ (*cf* section 1.4.1.4 page 36), on peut voir que la représentation GMDSDF ne spécifie aucun ordre d'exécution. On peut donc choisir les données entrantes sur la dimension temporelle ou sur la dimension verticale. Bien sûr le choix est ici évident, mais ce n'est pas toujours le cas, surtout si le nombre de dimensions est supérieur à deux. Les spécifications sur l'ordre d'exécution ne sont pas nécessaires dans le cadre de flux mono-dimensionnel mais elles s'avèrent être indispensables lorsque le flux est multi-dimensionnel.

4.2.3 Conclusion

Nous venons dans cette section de projeter ARRAY-OL vers un modèle de calcul pipeline. Afin d'obtenir une projection « intelligente », nous avons transformé l'application à l'aide des outils que nous avons conçus au chapitre précédent. Pour l'instant cette projection n'est que théorique, nous allons donc la mettre en œuvre sur deux modèles de calculs concrets : SDF et KPN. Le choix de ces modèles de calcul n'est pas un hasard et ceci pour plusieurs raisons :

- SDF et KPN sont déjà utilisés pour simuler le type d'applications visées par ARRAY-OL ;
- SDF et KPN sont des modèles à flots de données et on espère donc profiter de notre projection sur le modèle pipeline ;
- SDF et KPN sont déterministes ce qui simplifie grandement la recherche d'éventuelles erreurs ;
- SDF et KPN gèrent la synchronisation des tâches automatiquement ;
- enfin la projection se fait de manière systématique comme nous allons le voir ci-dessous.

4.3 Projection d'ARRAY-OL sur SDF

Au premier abord le choix de projeter ARRAY-OL sur SDF peut paraître incongru. En effet nous avons décrit SDF dans la section 1.2.1 page 11 comme étant un modèle à flot de données

synchrones destiné à des applications mono-dimensionnelles. Il nous faut donc concilier l'aspect multidimensionnel d'ARRAY-OL et les limites de SDF dans ce domaine. Cette tâche semble difficile puisqu'il n'est pas possible de modéliser en SDF les contraintes multidimensionnelles des applications ARRAY-OL comme nous avons pu le voir dans la section 1.2.2.1 page 13, mais ce n'est pas l'aspect modélisation de SDF que nous allons utiliser ici. Notre but est d'exécuter à l'aide de SDF une application préalablement décrite en ARRAY-OL.

Il semblerait plus naturel d'utiliser MDSDF ou GMDSDF comme support puisqu'ils permettent tous les deux de modéliser des applications multidimensionnelles. Mais ces deux modèles n'existent que sous forme théorique et n'ont fait l'objet d'aucune implémentation. En revanche SDF a été entièrement implémenter dans PTOLEMY puisqu'il est possible avec cet outil de décrire et de simuler des applications.

Nous allons donc étudier le portage d'une application en ARRAY-OL en SDF, puis nous verrons l'implémentation que nous avons réalisé au niveau de PTOLEMY.

4.3.1 Analyse théorique de la projection

ARRAY-OL et SDF sont différents mais se basent tous les deux sur des graphes de tâches. Cependant ARRAY-OL dispose de composants qui ne trouvent pas leur place dans un graphe SDF. Nous allons donc étudier la transposition de ces composants dans une représentation SDF.

4.3.1.1 Le modèle global

ARRAY-OL comporte deux niveaux de description : le modèle local et le modèle global. L'utilité de celui-ci peut être remise en question puisqu'il ne fait que relier les entrées et les sorties des modèles locaux. On peut donc légitimement se demander s'il est utile de le représenter en SDF puisqu'on peut se contenter de représenter tous les modèles locaux à un même niveau et de les relier entre eux. Mais nous avons choisi de dévier le moins possible du modèle ARRAY-OL en conservant le modèle global. En outre, nous pensons que ce dernier aide à la compréhension de l'application en offrant une plus grande lisibilité.

Comme nous l'indiquons dans la section 1.3.2.1 page 20, le modèle global est un graphe orienté où les sommets sont des tableaux ou des tâches et où les arêtes indiquent le sens de l'exécution. Une représentation telle quelle en SDF n'est pas possible. En effet, les tableaux ne peuvent être représentés par des nœuds puisqu'en SDF chaque nœud du graphe doit produire et consommer des données. On décide alors de représenter chaque tâche par un nœud et de se servir des arêtes pour symboliser les tableaux en même temps que le sens d'exécution du graphe. Les nœuds du graphe ont donc une nature hiérarchique puisqu'ils sont amenés à contenir les modèles locaux. La non-représentation des tableaux évite l'introduction de nœuds inutiles mais réduit l'expressivité du modèle.

Il nous faut désormais spécifier le nombre de données consommées et produites par chaque nœud. Or en ARRAY-OL une tâche ne manipule qu'un seul tableau à la fois sur chacune de ses arêtes entrantes ou sortantes. On fixe donc à un le nombre de données sur chaque arête dans notre graphe SDF.

Nous venons de transcrire le modèle global en ARRAY-OL, il nous faut maintenant nous occuper du modèle local.

4.3.1.2 Le modèle local

Le modèle local définit l'interaction entre une tâche et ses tableaux opérandes et résultats. Il est défini sous forme d'un graphe où chaque tableau est relié à la tâche de ce modèle. La représentation en SDF est donc immédiate, elle ne présente pas à ce stade de difficulté. Nous avons donc un premier graphe qui représente le modèle global en reliant des tâches entre elles et pour chacune de ces tâches nous avons un nouveau graphe qui représente le modèle local.

Mais le modèle local contient également des *tilers* qui permettent la manipulation des motifs, on peut d'ailleurs distinguer les *tilers* d'entrée qui vont servir à découper les tableaux et à construire les motifs et les *tilers* de sortie qui vont regrouper des motifs pour constituer des tableaux. Notre problème est donc de savoir comment représenter ces *tilers* ? Ils sont « actifs » car ils agissent sur la structure des données dans le flux et donc on ne peut pas les éliminer à l'inverse des tableaux dans le modèle global. Comme il n'existe aucun autre moyen, nous sommes obligés de les représenter comme des tâches en utilisant des nœuds du graphe. Mais cette solution n'est pas parfaite puisqu'elle nous oblige à représenter à un même niveau de description les traitements sur la structure des données et les traitements sur les données elles mêmes.

Au final, la projection du modèle local en SDF comporte un nœud pour la tâche et un nœud par arête entrante et sortante afin de représenter les *tilers*. Il nous reste à déterminer le nombre de données consommées et produites par chaque nœud. Certaines valeurs sont faciles à trouver. Ainsi le modèle local consomme et produit nécessairement un tableau à chaque fois comme nous l'avons fixé dans le modèle global. Les *tilers* d'entrées vont donc consommer chacun une donnée, de même les *tilers* de sorties vont en produire également une. Le nœud représentant la tâche consomme un motif par arête entrante et en produit un par arête sortante. Mais il nous faut alimenter suffisamment ce nœud pour qu'il consomme et produise tous les motifs. Le nombre de données produites par les *tilers* d'entrée et le nombre de données consommées par les *tilers* de sortie doit donc être égal au nombre total de motifs qui est lui même égal au nombre total d'itérations de pavage[1] à savoir $\prod_{i=0}^{n-1} Q_i$. La figure 4.1 représente un exemple de modèle local en SDF. Cependant ce calcul soulève un problème, en effet si le tableau manipulé par la tâche s'avère être infini alors le nombre d'itérations de pavage risque lui aussi de l'être ce qui n'est pas possible en SDF. De plus l'utilisation de tableaux infinis comme *token* n'est pas réalisable, il nous faut donc restreindre le modèle aux tableaux bornés.

Finalement, nous avons réussi à projeter de l'ARRAY-OL sur SDF. L'aspect multidimensionnel est entièrement caché au niveau des *tilers* qui fournissent les motifs en fonction de leur matrice de pavage et d'ajustage. Nous pouvons donc bénéficier du modèle d'exécution présent dans SDF. Mais l'ordonnancement qui va découler de notre projection est il réellement intéressant ?

4.3.1.3 Ordonnancement

Nous pouvons remarquer que nous perdons l'expression du data-parallélisme au niveau des modèles locaux. Par construction nous imposons que les *tilers* d'entrées d'un modèle local soit exécutés m fois pour pouvoir exécuter la tâche. De même, nous imposons que

[1] Pour mémoire, les bornes d'itérations de pavage sont calculées globalement pour une la tâche puisque cette dernière consomme et produit le même nombre de motifs (*cf* section 1.3.2.3 page 26).

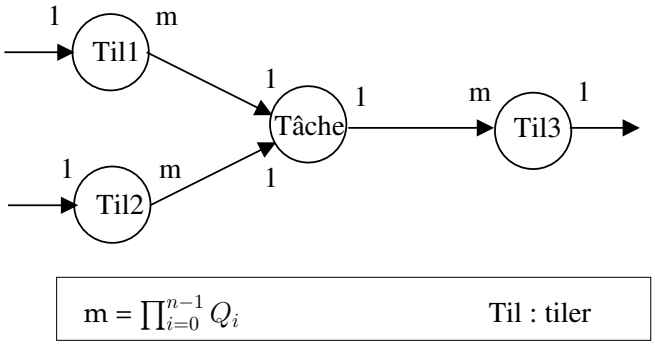


FIG. 4.1: Un modèle local en SDF

cette tâche soit exécutée m fois pour pouvoir exécuter les *tilers* de sortie. Il n'y aura donc pas d'exécution data-parallèle du modèle local. De plus nous accroissons inutilement les besoins en espace mémoire puisque tous les motifs produits par les *tilers* d'entrée doivent être stockés dans une file avant une exécution de la tâche. Cet inconvénient de la projection d'ARRAY-OL sur SDF est en fait proportionnel aux nombres de motifs. Plus il y a de motifs et plus la perte du data-parallélisme est dommageable.

Un autre problème intervient au niveau du modèle global. En effet, chaque tâche consomme un tableau pour en produire un autre, l'exécution d'une application va donc constituer en l'exécution séquentielle de ses tâches. Seule une application transformée pour mettre en avant un flot de tableaux tel que nous le décrivons dans la section 4.2.2.2 page 101 profite de l'aspect flot de données de SDF et de l'effet pipeline qui lui est associé. Ainsi le modèle global se retrouve soumis à un flot de tableaux, chaque tâche consomme un tableau sur le flux dès que la précédente a fini de calculer ce dernier. De manière schématique, on peut dire que les tâches ne sont plus exécutées une fois sur de « gros » tableaux mais plusieurs fois sur des « petits » ce qui a pour conséquence d'accélérer la production de résultats. De plus comme nous l'avons montré, nous pouvons également permettre l'utilisation de tableaux infinis. Ces derniers sont en effet limités au modèle global du plus haut niveau de hiérarchie et peuvent être facilement remplacés par une tâche pour simuler le remplissage du flux. La projection d'une application ARRAY-OL sur un modèle de calcul pipeline s'avère être en parfaite adéquation avec SDF.

4.3.2 Implémentation dans PTOLEMY

4.3.2.1 Présentation de PTOLEMY

Après avoir terminé l'étude théorique de la projection d'applications ARRAY-OL sur un modèle de calcul SDF, nous avons décidé de bénéficier des implémentations existantes du modèle SDF pour simuler nos applications ARRAY-OL. Le choix de l'implémentation a été rapide, nous nous sommes tout de suite intéressés à PTOLEMY **[**40**]** : l'environnement de modélisation et de simulation d'applications dédié aux systèmes embarqués développé par Edward Lee, le créateur de SDF.

D'un point de vue pratique PTOLEMY se présente sous la forme d'un logiciel écrit en JAVA disposant d'une interface graphique nommé VERGIL. Il est distribué sous licence BSD, ce qui autorise donc la modification de ses sources.

Mais PTOLEMY n'est pas restreint au modèle SDF, il permet d'utiliser plusieurs autres modèles de calculs. Lee définit PTOLEMY de la manière suivante : « PTOLEMY *est une collection d'entités et de relations entre ces entités. Certaines de ces entités, appelées acteurs, ont des fonctionnalités et peuvent communiquer grâce aux relations. D'autres entités, appelées domaines, impose un modèle de calcul sur les interactions entre les entités. Il y a plusieurs modèles de calculs dont Synchronous DataFlow (SDF), Process Network (PN), Discret Event (DE), etc.* ». PTOLEMY propose donc de modéliser des applications sous la forme d'un graphe d'acteurs et de les simuler en choisissant un modèle de calcul représenté par un domaine. Ces applications peuvent comporter plusieurs niveaux de hiérarchies avec un modèle de calcul propre à chacun de ces niveaux. On parle alors de simulation hétérogène.

Afin de pouvoir modéliser le plus grand nombre d'applications possibles, PTOLEMY propose de nombreux acteurs et domaines. Certains de ces acteurs fonctionnent avec tous les domaines alors que d'autres sont propres à certains domaines, certains sont relativement simples comme les opérations mathématiques élémentaires (addition, soustraction, etc.), d'autres sont des opérations très spécialisées comme des filtres IIR ou FIR. Mais la force de PTOLEMY réside principalement dans son grand nombre de modèles de calcul. On peut citer notamment hormis SDF : DE *(Discrete-event)*, CT (*Continious Time*) PN *(Process Network)*, FSM*(Finite State Machine)* et CSP *(Communicating Sequential Processes)*.

4.3.2.2 Implémentation

La représentation d'un modèle global est immédiate et ne nous a demandé aucun effort d'implémentation puisqu'il suffit à l'utilisateur de réaliser un graphe de tâches hiérarchiques et de lui associer un directeur SDF. Un directeur est un acteur spécial qui gère l'exécution en fonction du modèle de calcul qu'il représente. Une tâche hiérarchique est un acteur déjà présent dans PTOLEMY utilisé pour créer des niveaux de hiérarchies. Chacune de ces tâches est destinée à contenir un modèle local. Toutefois les tableaux manipulés par ARRAY-OL pouvant être n-dimensionnel, nous avons dû ajouter un nouveau type de *tokens* capables de les manipuler. La figure 4.2 représente un modèle global dans PTOLEMY.

La représentation d'un modèle local a nécessité la création de deux nouveaux acteurs : un acteur pour les *tilers* d'entrée et un pour les *tilers* de sortie. Le rôle de ces acteurs est bien sûr de découper les tableaux en motifs ou inversement de grouper les motifs pour faire des tableaux. Ils sont configurables et il est possible de saisir les différentes matrices à travers une interface graphique (*cf* figure 4.3). Nous nous sommes aperçus alors que nous donnons par le biais des *tilers* toutes les informations nécessaires à la modélisation d'une tâche ARRAY-OL. Nous pouvons donc faire de PTOLEMY, non plus un simple environnement de simulation mais un environnement de modélisation complet. Nous devons pour cela être en mesure de calculer les bornes de pavage. Si on reprend le calcul de ces bornes tel qu'il est décrit à la section 1.3.2.3 page 26, on voit alors qu'il faut déterminer quelle est la demi-tâche maîtresse, calculer les bornes grâce à ses vecteurs de pavage et enfin transmettre ces bornes à tous les *tilers* du modèle local. La plus grosse difficulté se situe au niveau de la transmission des bornes, il est en effet très difficile de communiquer entre acteurs en dehors du cadre normal des échanges de *tokens*. Nous avons décidé de profiter des spécifités des directeurs qui peuvent eux communiquer facilement avec tous les acteurs. Nous avons alors écrit notre propre directeur qui hérite directement du directeur SDF, mais qui récupère au niveau du *tiler* de la demi-tâche maîtresse les bornes de pavage avant de les communiquer à tous les autres *tilers* et ceci dans la phase dite d'« initialisation » qui précède la simulation proprement

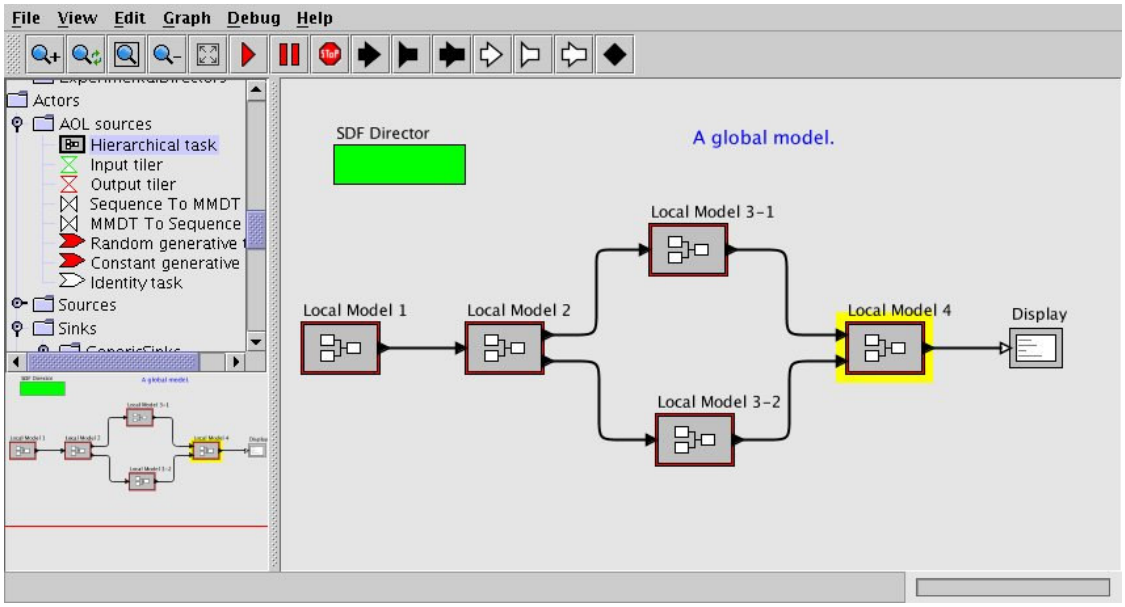


FIG. 4.2: Représentation d'un modèle global dans PTOLEMY

dite. La demi-tâche maîtresse est désignée par l'utilisateur au niveau de la configuration des *tilers*.

La représentation d'un modèle local ne nécessite pas d'autres modifications de PTOLEMY, nous avons cependant ajouté des acteurs appelés « MMDT to Sequence » et « Sequence to MMDT » qui servent de convertisseur entre nos *tokens* multidimensionnels et les *tokens* traditionnels manipulés par les acteurs de PTOLEMY et ceci afin de pouvoir réutiliser les acteurs de base comme les additions ou les multiplications. On peut également noter que si la tâche du modèle local est hiérarchique, il suffit d'utiliser les même acteurs que pour le modèle global.

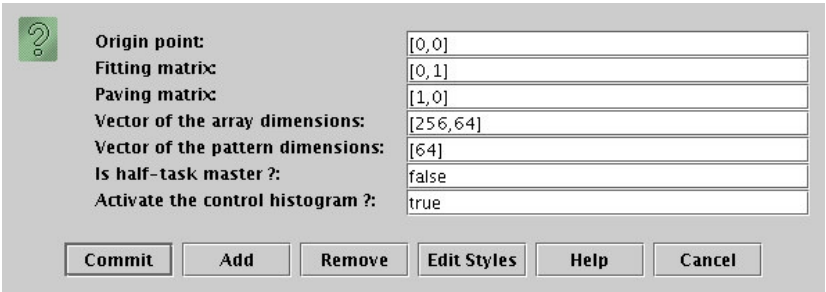


FIG. 4.3: Paramétrage d'un *tiler* dans PTOLEMY

La figure 4.4 représente un modèle local dans PTOLEMY, il est équivalent au modèle local présenté sur la figure 4.1 page 105. On peut notamment voir sur cette figure les *tilers* d'entrée et de sortie et notre directeur ARRAY-OL.

Nous avons donc réussi à faire de PTOLEMY un environnement de modélisation et de

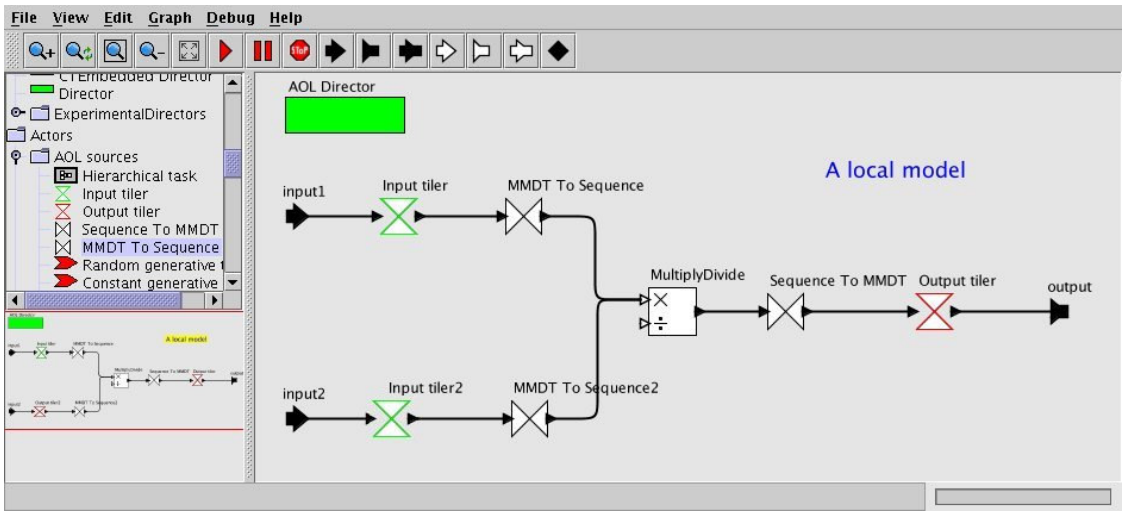


FIG. 4.4: Représentation d'un modèle local dans PTOLEMY

simulation d'applications ARRAY-OL. Je me dois de rajouter que cette implémentation a été en partie réalisée grâce au travail de plusieurs étudiants.

4.4 Projection d'ARRAY-OL sur les KPN

4.4.1 Présentation des KPN

Le modèle des réseaux de processus de Kahn *(Kahn Process Network ou* KPN*)* a été proposé par Kahn et MacQueen [33, 34] dans le but d'exprimer facilement des applications concurrentes. Dans ce modèle, différents processus s'exécutent indépendamments les uns des autres. Ces processus communiquent par des FIFOs unidirectionnelles, chacune de ses FIFOs a un et un seul processus producteur et a au plus un processus consommateur. Chaque donnée d'une FIFO est lue et produite une seule fois. La lecture dans une FIFO vide est bloquante, mais l'écriture n'est jamais bloquante : les FIFOs sont considérées comme non bornées. Les KPN sont déterministes : le nombre et la valeur des données produites ne dépendent que de la définition du réseau et jamais de l'ordonnancement.

Le calcul de ce dernier détermine la taille des FIFOs et la terminaison éventuelle de l'application. L'ordonnancement doit répondre aux deux impératifs suivants :

1. l'application doit s'exécuter complètement, ce qui sous entend qu'elle doit s'exécuter indéfiniment si il n'y a pas de terminaison possible ;
2. l'utilisation de FIFOs de tailles infinies n'étant pas possible de manière concrète, les FIFOs sont forcément bornées.

Malheureusement, Buck [10] a prouvé l'indécidabilité de ces deux conditions sur les graphes à flot de données booléennes qui sont des cas particuliers des réseaux de processus. Par extension elles sont également indécidables pour tous les réseaux de processus. Parks [54] a étudié les problèmes d'ordonnancement des KPN, il compare trois classes différentes d'ordonnancement dynamique : *data-driven*, *demand-driven* et une combinaison des deux. L'ordonnancement *data-driven* respecte la première condition, mais pas toujours la deuxième.

L'ordonnancement *demand-driven* peut provoquer des inter-blocages artificiels. La combinaison des deux proposée par Park autorise une exécution complète et bornée des réseaux de processus mais uniquement lorsque cela est possible.

On peut remarquer que les principes des KPN sont très proches de ceux de SDF, c'est effectivement exact puisque le modèle SDF est un cas particulier du modèle KPN.

4.4.2 Étude de la projection

4.4.2.1 Établissement de la projection

La projection d'ARRAY-OL sur KPN n'est pas identique à celle sur SDF. En effet, en SDF nous avions décidé de garder la distinction entre le modèle global et le modèle local en effectuant deux projections différentes et en utilisant une construction hiérarchique pour les relier. Nous pensions alors que la présence à un même niveau de toutes les tâches et de tous les *tilers* allait considérablement complexifier la lisibilité de l'application. Ici le problème est tout autre, en effet la consommation et la production des données sont gérées uniquement au niveau des processus sans que l'on ait à l'écrire explicitement lors de la spécification de l'application. La représentation des *tilers* devient donc inutile, on peut considérer que tous les mécanismes de consommation et de production sont pris en charge par les processus. On décide donc de modéliser à un même niveau les modèles locaux et globaux. Les *tilers* et les tâches d'un même modèle local sont alors représentés dans un seul et même processus. Les processus sont reliés entre eux grâce aux FIFOs en suivant les indications du modèle global comme on peut le voir la figure 4.5 .

Tous les niveaux de hiérarchie sont eux aussi représentés à un seul et même niveau. Il suffit juste d'insérer le réseau de processus représentant le deuxième niveau de hiérarchie à l'emplacement du processus représentant la tâche hiérarchique comme le montre la figure 4.5 . Cependant notre non-représentation des *tilers* introduit un problème. Dans une hiérarchie ARRAY-OL, la tâche hiérarchique consomme et produit des tableaux alors que le niveau de hiérarchie inférieure manipule des motifs qui sont des morceaux de ces tableaux et ce sont les *tilers* de la tâche qui établissent les liens entre les deux. Mais le processus contenant ces *tilers* n'est pas présent puisqu'il est remplacé par le réseau représentant la hiérarchie. Il nous faut donc trouver une représentation pour ces *tilers*. On décide alors d'introduire deux processus ayant pour seul rôle de faire le travail des *tilers* de la tâche hiérarchique. On place ces deux processus avant et après le début de la hiérarchie comme le montre la figure 4.5 .

En revanche de façon analogue à SDF, on projette les tableaux directement sur les FIFOs. Les tableaux avec une dimension infinie ne posent plus de problèmes, il suffit juste de les linéariser puisque les FIFOs sont elles mêmes infinies.

4.4.2.2 Exécution

De manière analogue à SDF, l'exécution d'une application ARRAY-OL non transformée est problématique. En effet, la possibilité d'utiliser des FIFOs de taille infinie pour stocker les tableaux ne change rien aux problèmes d'ordre d'exécution qui peut encore entraîner l'exécution infinie de processus. De plus, l'utilisation de l'aspect flot de données par ARRAY-OL est limitée puisque le résultat de ce qui est produit par un processus n'est généralement pas directement consommable par le processus suivant. Une application transformée ne pose pas ce genre de problèmes : l'ordre d'exécution est fixé et un processus peut consommer directement les résultats du processus précédent.

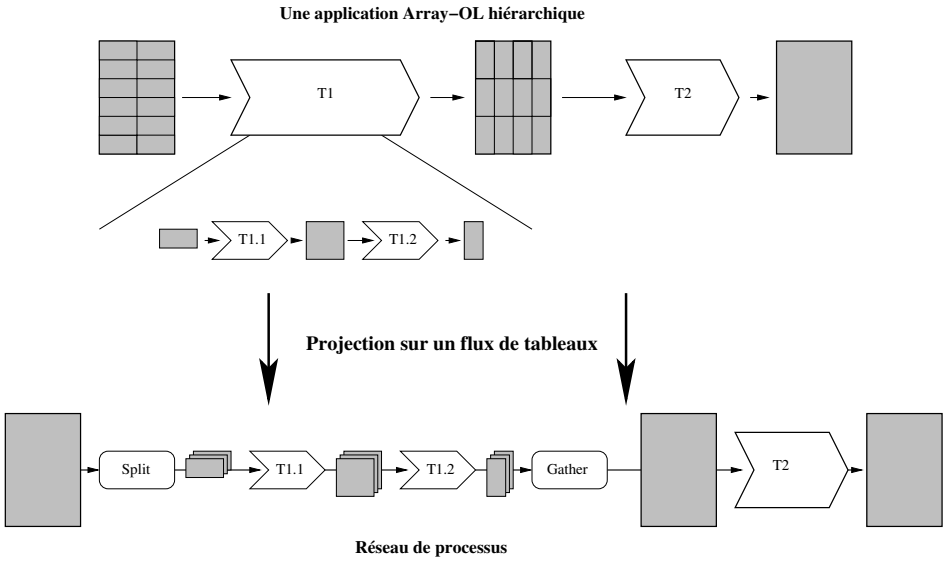


FIG. 4.5: Projection d'une application ARRAY-OL sur KPN.

Si on compare entre elles, les exécutions sur SDF et sur KPN, on constate que l'exécution sur KPN comporte deux avantages importants : d'une part, les processus peuvent être exécutés de manière parallèle contrairement à SDF et d'autre part, il est possible de faire une exécution distribuée comme l'a montré Abdelkader Amar dans sa thèse.

4.4.3 Implémentation

Les KPN sont très génériques, il existe plusieurs implémentations dans différents domaines : pour la modélisation hétérogène avec PTOLEMY **[**40**]**, pour la modélisation d'applications de traitement de signal et l'étude de leurs performances avec Yapi **[**19**]** et pour le métacalcul dans le domaine des systèmes d'information géographique avec Jade/PAGIS **[**61, 62**]**. Seule l'implémentation de Jade/PAGIS est distribuée, celles de PtolemyII et de Yapi utilisent des fils d'exécution pour représenter les différents processus.

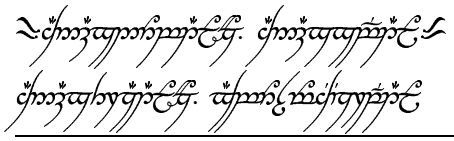
L'implémentation des réseaux de processus qui a servi à tester la projection d'ARRAY-OL sur KPN a été proposée par Abdelkader Amar **[**4, 3**]**. Elle a pour spécifité d'être distribuée et de reposer sur CORBA. Un des buts de cette implémentation est de cacher au développeur la complexité de mise en place d'une application distribuée. Ce dernier doit juste écrire la partie traitement des données de chaque processus et donner les prototypes de ces traitements. Un générateur de code est alors en mesure de produire un code pour une exécution distribuée cachant ainsi les détails de l'implémentation et de la synchronisation. Les processus distribués sont exécutés de manière data-parallèle.

La projection est évidente puisqu'elle repose sur l'implémentation des réseaux de processus d'Abdelkader Amar et que les *tilers* sont directement intégrés aux processus. Cette projection a été testée et les résultats de l'exécution d'une application ARRAY-OL sont longuement détaillés dans la thèse d'Abdelkader Amar **[**3**]**.

4.5 Conclusion

Nous venons d'étudier dans ce chapitre la projection d'ARRAY-OL vers différents modèles de calculs. Nous avons vu que l'utilisation des transformations décrites au chapitre précédent était nécessaire à une exécution « intelligente » d'ARRAY-OL. Nous avons également vu qu'ARRAY-OL était particulièrement bien adapté aux modèles à flot de données et nous avons proposé une implémentation dans le logiciel PTOLEMY. Or les modèles à flot de données sont déja utilisés pour simuler les applications du traitement du signal intensif, on obtient donc la confirmation de l'adéquation d'ARRAY-OL aux applications qu'il vise.

Conclusion



J.R.R. Tolkien, *Le Seigneur des anneaux*.

Récapitulation

Dans cette thèse, nous avons abordé la problématique de la modélisation des applications de traitement du signal systématique. Nous avons vu qu'une représentation par flux de données synchrones multidimensionnelles était particulièrement bien adaptée. Nous avons alors comparé plusieurs modèles différents dont ARRAY-OL qui a semblé être le plus apte à exprimer l'aspect multidimensionnel des données.

Afin de bénéficier de l'expression des dépendances présentes dans ARRAY-OL, nous avons étudié, dans le deuxième chapitre, différentes méthodes pouvant mener à l'optimisation d'une application. Nous avons justifié le choix du formalisme ODT en analysant les possibilités offertes par les techniques de transformations de boucles et en montrant que les ODT étaient en mesure d'exprimer parfaitement les dépendances de données. Puis, nous avons présenté un certain nombre de résultats issus de calculs sur ces ODT et qui nous ont servi dans l'établissement des optimisations.

Nous avons présenté dans le troisième chapitre cinq transformations : la fusion, le changement de pavage, l'aplatissement, la mise à niveau et le tiling ; nous avons étudié leurs principes et leurs utilisations. Elles constituent une « boîte à outils » capable d'effectuer de simples modifications ou des optimisations plus complexes sur des applications ARRAY-OL, mais surtout elles ne changent pas la nature de ces applications qui restent toujours des applications ARRAY-OL.

Enfin dans le quatrième chapitre, nous avons analysé la projection d'ARRAY-OL vers des modèles de calculs. Nous avons vu que l'utilisation des transformations décrites au chapitre précédent était nécessaire à une exécution « intelligente » d'ARRAY-OL. Nous avons également proposé une implémentation de la simulation d'ARRAY-OL dans le logiciel PTOLEMY. Enfin, notre analyse de la projection a montré l'adéquation existante entre ARRAY-OL et les modèles de calculs SDF et KPN.

Contribution

Au cours de ma thèse, j'ai essayé de réaliser un environnement complet de modélisation, d'optimisation et de simulation d'applications ARRAY-OL. Mon principal travail a bien sûr été de concevoir une « boîte à outils » de transformations qui permettent de réaliser les optimisations. Mais je me suis également intéressé à l'exécution d'ARRAY-OL afin de valider ces transformations et cela m'a conduit à étudier la modélisation des applications.

Optimisation des applications ARRAY-OL

J'ai tout d'abord repris les travaux de Julien Soula pour écrire une nouvelle démonstration claire et détaillée de l'optimisation appelée fusion. Cette démonstration m'a permis de réaliser une implémentation qui soit à la fois capable d'effectuer cette fusion et en mesure de servir de base à l'implémentation d'autres optimisations. J'ai complété ce travail en décrivant toutes les fusions sortant du cadre de deux tâches avec un tableau commun.

J'ai ensuite proposé une nouvelle version de la transformation appelée changement de pavage et qui n'avait été que rapidement abordée par Julien Soula. Il y a finalement deux changements de pavage : le changement de pavage par ajout de dimensions qui apporte une solution au problème de granularité des hiérarchies et le changement de pavage par agrandissement linéaire qui réduit les recalculs au sein d'une hiérarchie tout en modifiant également la granularité.

Enfin, le tiling est une transformation que j'ai introduite pour autoriser la création d'une hiérarchie à partir d'une seule tâche.

Finalement, la combinaison de ces transformations finit de constituer notre « boîte à outils » qui est désormais complète.

Modélisation et simulation d'applications

Les travaux décrits ci-dessous s'inscrivent naturellement dans la droite ligne de ceux sur l'optimisation, mais ils sont aussi le fruit d'une volonté qui a été mienne de valoriser ARRAY-OL en élaborant plusieurs outils pour la conception et la simulation d'applications.

J'ai commencé par chercher des langages similaires à ARRAY-OL, or malgré la relative ancienneté des travaux de Lee et de Murthy, aucun document traitant d'ARRAY-OL ne faisait référence à GMDSDF. J'ai alors écrit une comparaison détaillée de ces deux langages, ce qui m'a ensuite donné l'idée de réaliser la projection d'ARRAY-OL sur SDF. Cette projection et l'implémentation qui en découle répondent à trois objectifs :

- valider les optimisations et montrer leur importance pour l'obtention d'une simulation « intelligente » ;
- fournir avec PTOLEMY un environnement de modélisation et de simulation qui soit à la fois simple, complet et connu de la communauté du traitement du signal ;
- bénéficier de la simulation hétérogène de PTOLEMY pour simuler conjointement les parties de TSS et de TDI de nos applications.

Enfin, afin de faciliter l'apprentissage d'ARRAY-OL et la modélisation des applications, j'ai proposé et encadré plusieurs stages étudiants qui ont abouti à la réalisation d'« ARRAY-OL exemple ». Ce logiciel est devenu un outil incontournable pour l'utilisation d'ARRAY-OL.

Analyse critique

Mon utilisation quasi quotidienne d'ARRAY-OL, les nombreuses présentations que j'en ai fait auprès d'autres doctorants ou auprès des étudiants et les divers calculs que j'ai effectué avec les ODT m'ont amené à émettre plusieurs critiques.

De la complexité d'ARRAY-OL

J'ai constaté que certains principes d'ARRAY-OL restent souvent relativement obscurs et ceci même pour des gens qui ont déjà travaillé dessus. ARRAY-OL est difficile d'accès ; certains de ses aspects, comme le fait que les motifs puissent avoir plus de dimensions que les tableaux dont ils sont extraits, sont mal compris et ils sont donc généralement occultés. Ainsi, les logiciels développés chez THALES sont souvent réduits à un sous ensemble d'ARRAY-OL où les motifs sont limités à leurs boîtes englobantes, où les matrices de pavage et d'ajustage ne contiennent que des vecteurs parallèles aux axes et où l'usage des dimensions toriques est proscrite.

Autre inconvénient d'ARRAY-OL, le calcul des différentes matrices est une tâche longue et fastidieuse. Comme nous l'avons vu dans le premier chapitre, le nombre de matrices à calculer est égal à dix fois le nombre d'arêtes du graphe. L'utilisation d'un outil graphique tel « ARRAY-OL exemple » est donc indispensable.

Tous ces défauts d'ARRAY-OL sont structurels, il est difficile d'y remédier sans avoir à modifier le modèle lui même. Toutefois, nous pouvons noter que même si toutes les subtilités d'ARRAY-OL ne sont pas immédiatement assimilables, ce dernier reste utilisable.

De l'inadaptation des ODT

Nous avons vu que les ODT sont un excellent moyen pour représenter les dépendances de données au sein d'un modèle local ARRAY-OL, mais nous avons également vu qu'une représentation ODT ne correspondait pas exactement à une application ARRAY-OL. Les calculs sur les ODT nous garantissent uniquement que les dépendances de données restent correctes, le résultat n'est pas nécessairement valide du point de vue ARRAY-OL. Il faut donc garder en tête une représentation ARRAY-OL de l'expression ODT que l'on manipule. On peut notamment citer deux exemples particulièrement révélateurs :

1. dans le cadre de la fusion, l'introduction de recalculs n'apparaît pas dans la forme ODT des résultats ;
2. si on juxtapose la représentation ODT d'une tâche ayant des motifs disjoints, comme sur la figure 3.16 page 84, avant et après un changement de pavage par ajout de dimensions, alors il n'est pas possible de voir que les deux représentations ne sont pas équivalentes et que le changement de pavage a introduit des erreurs.

Un autre exemple, lui aussi intéressant, est donné dans la section B.4.2 page 131, mais il est trop long et trop complexe pour être détaillé ici. Le pouvoir d'expressivité des ODT est donc très limité et nombre des difficultés que j'ai rencontrées dans l'élaboration des transformations proviennent de cette limitation.

Enfin, les ODT sont incapables d'exprimer les dépendances entre les itérations de pavage, il n'est donc pas possible de tester des équivalents des états et des délais de GMDSDF au niveau des ODT.

Perspectives

Amélioration des développements existants

Plusieurs développements ont été réalisés au cours de cette thèse, une première perspective est bien sûr de maintenir et de poursuivre ces développements.

« ARRAY-OL exemple » vient de bénéficier de nombreuses améliorations lors du récent stage de Sylvain Buisine. Il est désormais possible de définir et de visualiser simultanément tous les *tilers* d'un même modèle local. Cependant je pense qu'« ARRAY-OL exemple » peut encore être amélioré notamment en l'interfaçant avec un environnement de modélisation comme PTOLEMY.

Concernant le support d'ARRAY-OL dans PTOLEMY, un travail de maintenance est indispensable pour adapter nos développements à la sortie des dernières versions de PTOLEMY. Ainsi, il faut déjà mettre à jour l'implémentation actuelle pour la nouvelle version de PTOLEMY qui est sortie en juillet 2005. Parallèlement, il est envisageable d'ajouter l'implémentation des transformations afin de disposer d'une plate forme complète dédiée à ARRAY-OL.

L'implémentation des transformations est à compléter en ajoutant le support des transformations autre que la fusion. Mais contrairement à ce qu'on pourrait croire, ce n'est pas un travail très complexe. En effet, le support des ODT existe déjà et la passerelle entre ARRAY-OL et les ODT est également présente, il n'y a donc pas de difficultés majeures.

Extension d'ARRAY-OL

Lorsque nous avons comparé ARRAY-OL et GMDSDF, il est apparu que l'atout principal de GMDSDF était le support des états et des délais qui permettent de modéliser des tâches telles que des accumulateurs. Il est primordial d'ajouter cette fonctionnalité dans ARRAY-OL. Le problème ne se situe évidemment pas au niveau de la modélisation, mais c'est au niveau de l'optimisation qu'on rencontre des difficultés. En effet, comme nous l'avons vu précédemment, les ODT sont incapables d'exprimer les dépendances entre les itérations d'un même modèle local. Il n'est donc pas possible d'étudier l'impact de l'ajout des états et des délais à l'aide de ce formalisme, il faudra donc trouver une solution différente.

Une autre extension possible est actuellement étudié par Ouassila Labbani qui est une doctorante de l'équipe WEST. En effet, elle a effectué une analyse pertinente **[**38**]** de l'ajout de flots de contrôle dans ARRAY-OL. Ces flots de contrôle sont inspirés du domaine des systèmes réactifs synchrones et sont d'autant plus intéressants qu'ils conservent l'aspect déterministe d'ARRAY-OL. De plus, Ouassila Labbani prévoit de poursuivre ces travaux en essayant de compiler des applications ARRAY-OL vers des langages synchrones de type LUSTRE.

Pilotage des transformations

Nous avons décrit dans cette thèse une « boîte à outils » de transformations destinées à ARRAY-OL. Mais ces transformations ne sont intéressantes que si elles sont utilisées intelligemment. Le choix d'appliquer telle ou telle transformation dépend de l'architecture ciblée et de la nature même de l'application. Une application qui consomme de grandes quantités de mémoire est bien sûr amener à être transformée et ceci quelque soit l'architecture sur laquelle elle doit être exécutée. Mais dans le cadre de l'embarqué, il faudra combiner la fusion, l'aplatissement et le changement de pavage pour ajuster finement la quantité de mémoire

requise lors de l'exécution. On peut également utiliser le changement de pavage et le tiling pour gérer le nombre d'itérations de pavage et pour les faire correspondre, par exemple, au nombre d'unités de calcul. On peut envisager d'appliquer la mise à niveau sur des parties distinctes de l'application et délimiter ainsi des séquences destinées à être exécutées de manière data-parallèle. Les différents cas que l'on vient de citer ne sont que de simples exemples et une étude approfondie du sujet est donc nécessaire.

Bibliographie

[1] M. D. Adams. The JPEG-2000 still image compression standard. Technical Report N2412, ISO/IEC JTC 1/SC 29/WG 1, septembre 2001. http://www.jpeg.org/wg1n2412.pdf.

[2] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4) :491–542, 1987.

[3] A. Amar. *Support d'exécution pour le metacomputing à l'aide de CORBA*. Thèse de doctorat, USTL, LIFL, décembre 2003.

[4] A. Amar, P. Boulet, J.-L. Dekeyser, and F. Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *ParCo'2003, Parallel Computing*, Dresden, Germany, septembre 2003.

[5] C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jourdan, and J. Mattioli. Automatic data mapping of signal processing applications. In *Application Spec. Array Processors*, pages 350–362, Zurich, Switzerland, juillet 1997.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4) :345–420, 1994.

[7] P. L. G. T. G. M. L. Borgne, and C. L. Marie. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9) :1321–1335, septembre 1991.

[8] P. Boulet, J.-L. Dekeyser, F. Devin, and P. Marquet. A visual development environment for meta-computing applications. In *HCI International 2001, 9th Int'l Conf. on Human-Computer Interaction*, volume 1, pages 983–987, New Orleans, LA, août 2001. Lawrence Erlbaum Associates, Publishers.

[9] P. Boulet, J.-L. Dekeyser, J.-L. Levaire, P. Marquet, J. Soula, and A. 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.

[10] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Thèse de doctorat, University of California at Berkeley, 1993.

[11] D. Callahan. *A global approach to detection of parallelism*. Thèse de doctorat, Rice University, mars 1987.

[12] S. Carr. *Memory-Hierarchy Management*. Thèse de doctorat, Rice University, 1993.

[13] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software - Practice and Experience*, 24(1) :51–77, 1994.

[14] P. Caspi and M. Pouzet. A functional extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, mai 1995. World Scientific.

[15] M. J. Chen and E. A. Lee. Design and implementation of a multidimensional synchronous dataflow environment. In *1995 Proc. IEEE Asilomar Conf. on Signal, Systems, and Computers*, 1995.

[16] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, 1995.

[17] A. Darte. De l'organisation des calculs dans les codes répétitifs. Habilitation à diriger des recherches, École Normale Supérieure de Lyon, décembre 1999.

[18] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9) :1175–1193, 2000.

[19] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI : Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, juin 2000. ACM Press.

[20] A. Demeure. Les ODT : propositions de notation pour décrire des opérateurs de distribution de tableaux. Rapport technique, Thomson Marconi Sonar, Sophia-Antipolis, France, 1998.

[21] A. Demeure, A. Lafarge, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL : Proposition d'un formalisme tableau pourle traitement de signal multidimensionnel. In *Gretsi*, Juan-Les-Pins, France, septembre 1995.

[22] F. Devin, P. Boulet, J.-L. Dekeyser, and P. Marquet. Gaspard : a visual data-parallel programming environment for signal processing applications. In *International Conference on Parallel Computing in Electrical Engineering, PARELEC'2002*, Warsaw, Poland, septembre 2002.

[23] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *International Conference on Fifth Generation Computer Systems*, pages 693–264, 1988.

[24] P. Dumont and P. Boulet. Transformations de code Array-OL : implémentation de la fusion de deux tâches. Rapport technique, Laboratoire d'Informatique fondamentale de Lille et Thales Communications, octobre 2003.

[25] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 21(1) :23–53, 1991.

[26] A. Fraboulet. *Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués*. Thèse de doctorat, INSA, Lyon, France, novembre 2001.

[27] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 281–295, New Haven, Conn., 1992. Berlin : Springer Verlag.

[28] G. R. Gao and V. Sarkar. Optimization of array accesses by collective loop transformations. In *International Conference on Superconputing*, pages 194–205, 1991.

[29] S. Girbal. *Optimisation d'applications - Composition de transformations : modèle et outils*. Thèse de doctorat, Université Paris XI, septembre 2005.

[30] M. Gordon. A stream compiler for communication-exposed architectures, mars 2002.

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, septembre 1991.

[32] P. V. Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3) :113–159, 1992.

[33] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, août 1974.

[34] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77 : Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.

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

[36] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin : Springer Verlag.

[37] D. Kulkarni and M. Stumm. Loop and Data Transformations : A tutorial. Internal document, a tutorial guide., juin 1993.

[38] O. Labbani, J.-L. Dekeyser, P. Boulet, and Éric Rutten. Introducing control in the gaspard2 data-parallel metamodel : Synchronous approach. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems*, Montego Bay, Jamaica, octobre 2005.

[39] E. A. Lee. Multidimensional streams rooted in dataflow. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, janvier 1993. North-Holland.

[40] E. A. Lee. *Overview of the Ptolemy Project*. University of California, Berkeley, mars 2001.

[41] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, janvier 1987.

[42] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proc. of the IEEE*, septembre 1987.

[43] A. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Languages and Compilers for Parallel Computing*, pages 92–106, 1994.

[44] R. Manduchi, G. M.Cortelazzo, and G. A. Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 1993.

[45] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2) :193–209, 1997.

[46] C. Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. Thèse de doctorat, Université de Rennes I, décembre 1989.

[47] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4) :424–453, juillet 1996.

[48] L. Morel. Efficient compilation of array iterators for lustre. *Electr. Notes Theor. Comput. Sci.*, 65(5), 2002.

[49] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. Thèse de doctorat, University of California, Berkeley, CA, 1996.

[50] P. K. Murthy and E. A. Lee. A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices. Rapport de recherche, Electronics Research Laboratory, mars 1995.

[51] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8) :2064–2079, août 2002.

[52] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Improving cache performance through tiling and data alignment. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 167–185, 1997.

[53] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2) :142–149, 1999.

[54] T. M. Parks. *Bounded Scheduling of Process Networks*. Thèse de doctorat, EECS Department, University of California, Berkeley, CA, décembre 1995.

[55] N. H. Pascal. A tutorial of lustre.

[56] F. Rocheteau and N. HALBWACHS. Pollux : A lustre based hardware design environment.

[57] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism, 1996.

[58] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat, LIFL, décembre 2001.

[59] J. Soula, P. Marquet, J.-L. Dekeyser, and A. Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, septembre 2001. Lecture Notes in Computer Science vol. 2127.

[60] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit : A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[61] D. Webb, A. Wendelborn, and K. Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, avril 1999.

[62] D. Webb, A. Wendelborn, and J. Vayssière. A study of computational reconfiguration in a process network. In *IDEA7*, Victor Harbour, South Australia, février 2000.

Annexe A

Notations, limitations et contraintes

A.1 Modèle global

- il ne peut y avoir de cycle dans le graphe du modèle global ;
- chaque tableau ne peut avoir qu'une seule dimension infinie.

A.2 Modèle local

- la numérotation des tableaux commence toujours à zéro ;
- les motifs sont forcément de taille finie ;
- assignation unique : les éléments des tableaux résultats doivent être calculés une et une seule fois ;
- toute tâche se doit d'avoir une « demi-tâche » résultat exacte ; sa partie résultat doit être telle que :
  - les origines des tableaux soient à zéro ;
  - les vecteurs de pavage et d'ajustage soient positifs et parallèles aux axes ;
  - l'utilisation du modulo soit interdite ;
- une tâche hiérarchique doit avoir le motif opérande de sa tâche supérieure comme tableau d'entrée de sa première sous-tâche. De même le tableau calculé par la dernière sous-tâche doit correspondre au motif produit par la tâche supérieure.

A.3 ODT

- si un motif a une de ses dimensions de taille 1, aucun ajustage n'apparaît pour cette dimension dans la forme ODT de la tâche concernée ;
- de façon identique pour un tableau avec une de ses dimensions de taille 1, aucun pavage n'apparaît (Il n'y a donc pas de bornes valant 1 dans le gabarit).

A.4 Fusion

- deux tâches, pour pouvoir être fusionnées, doivent avoir au moins un tableau commum que la première produit et que la deuxième consomme ;

– la tâche produisant ce tableau intermédiaire ($A2$) doit le faire par le biais d'une « demi-tâche » exacte.

Annexe B

Calcul de la fusion

Cet appendice présente une démonstration complète du processus de fusion en reprenant pas à pas la démarche décrite section 3.2.2 page 67. La version originelle de cette démonstration est présente dans la thèse de Julien Soula [58]. Comparée à celle de Julien Soula, cette version est beaucoup plus proche du formalisme ODT. En outre, une distinction a été faite tout au long du calcul entre les parties opérandes et résultats des opérateurs. Cette distinction apporte une plus grande clarté qui m'a permis de réaliser l'implémentation plus facilement.

B.1 Calcul des dépendances

B.1.1 Utilisation des propriétés des ODT exacts

– Nous partons donc avec notre expression intermédiaire à transformer :

$$\underbrace{\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} P_{res,1} & 0 & F_{res,1} \end{matrix}}}_{\text{ODT résultat exact}} . (M_2)_{\mathbf{M}} . (S_2)_{\mathbf{S}} . \begin{vmatrix} P_{op,2} & F_{op,2} & 0 \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– Grâce aux propriétés sur l'inversion des ODT exacts (*cf* 2.4.3.3 page 60) :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \left\lfloor \overline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} \right. . \begin{vmatrix} P'_{res,1} \\ 0 \\ F'_{res,1} \end{vmatrix} \underbrace{. (M_2)_{\mathbf{M}}}_{\text{modulo inutile}} . (S_2)_{\mathbf{S}} . \begin{vmatrix} P_{op,2} & F_{op,2} & 0 \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On utilise ensuite la suppression des modulos redondants (*cf* 2.4.3.3 page 60) :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \left\lfloor \overline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} \right. . \begin{vmatrix} P'_{res,1} \\ 0 \\ F'_{res,1} \end{vmatrix} . \underbrace{(S_2)_{\mathbf{S}}}_{\text{shift à déplacer}} . \begin{vmatrix} P_{op,2} & F_{op,2} & 0 \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On utilise la composition de la projection et du shift (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \left\lfloor \overline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} \right. . \begin{pmatrix} P'_{res,1} \times S_2 \\ 0 \\ F'_{res,1} \times S_2 \end{pmatrix}_{\mathbf{S}} . \underbrace{\begin{vmatrix} P'_{res,1} \\ 0 \\ F'_{res,1} \end{vmatrix} . \begin{vmatrix} P_{op,2} & F_{op,2} & 0 \end{vmatrix}}_{\text{projections à multiplier}} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– Enfin on se sert de la composition des projections (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\lfloor . \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} \underbrace{. \begin{pmatrix} P'_{res,1} \times S_2 \\ 0 \\ F'_{res,1} \times S_2 \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} P'_{res,1} \times P_{op,2} & P'_{res,1} \times F_{op,2} & 0 \\ 0 & 0 & 0 \\ F'_{res,1} \times P_{op,2} & F'_{res,1} \times F_{op,2} & 0 \end{vmatrix}}_{\text{variables à renommer}} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On renomme quelques variables :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\lfloor . \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} . \begin{pmatrix} S_p \\ 0 \\ S_f \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} P'' \\ 0 \\ F'' \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

B.1.2 Complétion des motifs

– Pour que notre expression réponde à nos désidérata, il faut que les points qu'elle génère remplissent toutes les dimensions d'ajustages D_1 et D_2. Si on examine les itérations sur notre gabarit, qui deviendra celui de notre nouvelle expression, il est facile de voir qu'on désigne tous les points de D_2. La segmentation, quant à elle, génère les points de $D_{op,1}$. En revanche on ne peut rien déterminer pour $D_{res,1}$. Pour compléter $D_{res,1}$, on supprime les dimensions existantes avec une projection et on les recrée avec une segmentation (*cf* 2.4.3.1 page 57). Puis on les limite grâce à un modulo.

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\lfloor . \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} . \begin{pmatrix} S_p \\ 0 \\ S_f \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} P'' \\ 0 \\ F'' \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}}$$
$$\subset \begin{pmatrix} \infty \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\begin{matrix} 1 & 0 & 0 \end{matrix}} . \underbrace{\begin{vmatrix} 1 & 0 & 0 \end{vmatrix}}_{\text{projection à déplacer}} . \begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\lfloor . \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}} . \begin{pmatrix} S_p \\ 0 \\ S_f \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} P'' \\ 0 \\ F'' \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}}$$

– La nouvelle projection élimine les dimensions d'ajustages avant qu'elles ne soient recrées par la nouvelle segmentation. Or on peut simplifier l'expression en déplaçant la nouvelle projection vers la droite (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} \infty \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\begin{matrix} 1 & 0 & 0 \end{matrix}} . (Q_1)_{\mathbf{M}} . \begin{vmatrix} 1 & 0 & 0 \end{vmatrix} . \underbrace{\lfloor . \overline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}}}_{\text{segmentation inutile}} . \begin{pmatrix} S_p \\ 0 \\ S_f \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} P'' \\ 0 \\ F'' \end{vmatrix} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On supprime $\left(\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{smallmatrix}\right)$ car cette segmentation ne servait qu'à créer l'ajustage opérande de $Q_1 D_1$ qui a désormais disparu (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} \infty \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\begin{matrix} 1 & 0 & 0 \end{matrix}} . (Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \underbrace{\begin{vmatrix} 1 & 0 & 0 \end{vmatrix} . \begin{vmatrix} P'' \\ 0 \\ F'' \end{vmatrix}}_{\text{projections à fusionner}} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On fusionne nos deux projections (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} \infty \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \overline{\underline{1 \quad 0 \quad 0}} \, (Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . |P''| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– Il ne nous reste plus qu'à déplacer notre segmentation vers la droite (*cf* 2.4.3.1 page 56) :

$$\begin{pmatrix} \infty \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} \begin{pmatrix} Q_1 \\ \infty \\ \infty \end{pmatrix}_{\mathbf{M}} . \overline{\underline{1 \quad 0 \quad 0}} . \lfloor . (S_p)_{\mathbf{S}} . |P''| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– On peut fusionner les deux modulos et continuer à déplacer notre segmentation. Cette dernière change de taille (*cf* 2.4.3.1 page 57). :

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \lfloor . \begin{pmatrix} S_p \\ 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \left| \begin{matrix} P'' & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \right| . \overline{\left| \begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{0} \end{matrix} \right|} . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} .$$

– Pour déplacer encore la segmentation, on est obligé d'agrandir le gabarit afin d'ajouter des contraintes sur les dimensions qui seront désormais créées avant. On peut fixer la valeur de ces contraintes en se basant sur celles du modulo car les dimensions créées au niveau de la segmentation ne sont pas modifiées jusqu'au modulo (*cf* 2.4.3.1 page 57).

$$\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}_{\mathbf{M}} . \lfloor . \begin{pmatrix} S_p \\ 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \left| \begin{matrix} P'' & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \right| . \left(\begin{matrix} Q_2 \\ D_{op,2} \\ D_{res,2} \\ \hline D_{op,1} \\ D_{res,1} \end{matrix} \right)_{\mathbf{G}} \overline{\left| \begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{0} \end{matrix} \right|} .$$

La séparation dans le gabarit signifie simplement que la partie supérieure de ce dernier s'applique à P'' dans la projection et à la matrice identité dans la segmentation.

– Nous avons donc réussi à obtenir une tâche ARRAY-OL, mais qui hélas comporte des parties fractionnaires à cause de l'utilisation de l'inverse de l'ODT exact. Pour supprimer ces valeurs fractionnaires, on décide de les isoler en ne gardant que le pavage (l'ajustage étant lui entier).

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . |P''| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{1 \quad 0 \quad 0}} .$$

B.2 Éliminations des parties fractionnaires

– On part de notre nouvelle expression :

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P'_{res,1} \times P_{op,2} \quad P'_{res,1} \times F_{op,2} \quad 0 \right| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{1 \quad 0 \quad 0}} .$$

– On renomme quelques variables.

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P_{\mathcal{M}} \quad F_{\mathcal{M}} \quad 0 \right| . \begin{pmatrix} Q_2 \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{array}{ccc} 1 & 0 & 0 \end{array}} \ .$$

Cette expression est en réalité la représentation d'une tâche ARRAY-OL entre les espaces d'itérations Q_1 et Q_2. Les itérations de pavage (respectivement d'ajustage) sont en fait les itérations de macro-pavage (macro-ajustage). On peut constater qu'à ce stade de notre raisonnement, la taille du macro-motif résultat est égale à la taille du motif résultat original (multiplication par la matrice identité). Rappelons que ce dernier est construit par l'ajustage spatial qu'on vient de mettre de côté pour simplifier notre expression.

B.2.1 Segmentation du gabarit de macro-pavage

– Pour supprimer les valeurs fractionnaires dans les vecteurs de macro-pavage $P_{\mathcal{M}}$ nous allons segmenter le gabarit de macro-pavage (*cf* 2.4.3.2 page 59). Cette opération a un double effet : elle réduit le nombre d'itérations de macro-pavage donc le nombre de calculs sur $T3$, mais elle augmente aussi la taille du macro-motif. De plus, l'apparition d'une segmentation explique la présence d'un macro-motif résultat.

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P_{\mathcal{M}} \quad F_{\mathcal{M}} \quad 0 \right| . \begin{vmatrix} \Lambda & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} . \begin{pmatrix} Q_2/\lambda \\ \lambda \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{array}{cccc} \Lambda & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}} . \overline{\begin{array}{ccc} 1 & 0 & 0 \end{array}} \ .$$

– On multiplie les deux projections et les deux segmentations.

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P_{\mathcal{M}} \times \Lambda \quad P_{\mathcal{M}} \quad F_{\mathcal{M}} \quad 0 \right| . \begin{pmatrix} Q_2/\lambda \\ \lambda \\ D_{op,2} \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{array}{cccc} \Lambda & 1 & 0 & 0 \end{array}} \ .$$

– λ appartient à la fois à l'ajustage opérande et à l'ajustage résultat, nous allons séparer de manière explicite les macro-ajustages opérandes et résultats car nous les avons traités comme ne faisant qu'un dans la dernière opération.

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P_{\mathcal{M}} \times \Lambda \quad P_{\mathcal{M}} \quad F_{\mathcal{M}} \quad 0 \quad 0 \right| . \begin{pmatrix} Q_2/\lambda \\ \lambda \\ D_{op,2} \\ \lambda \\ D_{res,2} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{array}{ccccc} \Lambda & 0 & 0 & 1 & 0 \end{array}} \ .$$

– On renomme quelques variables et on supprime $D_{res,2}$ qui est inutile :

$$(Q_1)_{\mathbf{M}} . \lfloor . (S_p)_{\mathbf{S}} . \left| P'_{\mathcal{M}} \quad P_{\mathcal{M}} \quad F_{\mathcal{M}} \quad 0 \right| . \begin{pmatrix} Q' \\ \lambda \\ D_{op,2} \\ \lambda \end{pmatrix}_{\mathbf{G}} . \overline{\begin{array}{cccc} \Lambda & 0 & 0 & 1 \end{array}} \ .$$

– Où en sommes nous ? Le macro-pavage n'a plus de parties fractionnaires, le macro-ajustage résultat n'en a pas non plus, il ne reste que le macro-ajustage opérande sur lequel nous allons travailler avec le shift.

$$\lfloor . (S_p)_{\mathbf{S}} . |P_{\mathcal{M}} \quad F_{\mathcal{M}}| . \begin{pmatrix} \lambda \\ D_{op,2} \end{pmatrix}_{\mathbf{G}} . \overline{0 \quad 0} \quad .$$

B.2.2 Vers le calcul d'une boîte englobante

Pour supprimer toutes les composantes fractionnaires du macro-ajustage nous allons calculer la boîte englobante des points générés. Mais afin que cette boîte ne soit pas trop grande (*cf* encadré page 59), nous allons réduire la taille des vecteurs de macro-ajustage en séparant les parties entières et fractionnaires. Mais cela nous oblige à doubler le nombre de vecteurs ce qui élève au carré le nombre d'itérations, il faut donc dans un premier temps limiter ce nombre.

B.2.2.1 Segmentation du gabarit de macro-ajustage

– On renomme quelques variables :

$$\lfloor . (S_p)_{\mathbf{S}} . |v| . (q)_{\mathbf{G}} . \underline{0} \quad .$$

– Nous effectuons une nouvelle fois une segmentation du gabarit (*cf* 2.4.3.2 page 59), ce qui a pour effet de réduire les bornes d'itérations, mais pas le nombre d'itérations. Nous donnons directement le résultat du calcul, puisque ce dernier est identique à celui effectué précédemment. La segmentation issue du calcul est absorbée par la segmentation nulle que nous n'avons pas mise pour simplifier la lecture.

$$\lfloor . (S_p)_{\mathbf{S}} . |v \times \Lambda' \quad v| . \begin{pmatrix} q/\lambda' \\ \lambda' \end{pmatrix}_{\mathbf{G}} .$$

B.2.2.2 Séparation des parties entières et fractionnaires.

Il ne reste des composantes fractionnaires que dans v. Nous allons donc utiliser l'inclusion suivante pour réduire la taille des vecteurs fractionnaires afin de ne pas calculer une boîte englobante trop grande.

$$\lfloor . (S_p)_{\mathbf{S}} . |v| . (\lambda')_{\mathbf{G}} \subset (S_p)_{\mathbf{S}} . |Ent(v) \quad Frac(v)| . \begin{pmatrix} \lambda' \\ \lambda' \end{pmatrix}_{\mathbf{G}} .$$

Cette nouvelle expression fait passer le nombre d'itérations de λ' à $\lambda' \times \lambda'$ d'où l'intérêt de la segmentation du gabarit d'ajustage qui avait réduit la taille des bornes.

B.2.2.3 Calcul de l'englobant de la partie fractionnaire.

Enfin on calcule la boîte englobante :

$$\lfloor . (S_p)_{\mathbf{S}} . |Frac(v)| . (\lambda')_{\mathbf{G}} \subset \left(\overrightarrow{min}\right)_{\mathbf{S}} . |1| . \left(\overrightarrow{max} - \overrightarrow{min}\right)_{\mathbf{G}} .$$

C'est fini, notre tâche est entière, l'arrondi a disparu, il faut désormais « remonter » les résultats obtenus.

B.3 Remontée des résultats

Aucune opération n'est effectuée, on replace juste les expressions à leur emplacement d'origine.

– étape 1 : (*cf* B.2.2.3)

$$\left(\overrightarrow{min}\right)_{\mathbf{S}} . \left|1\right| . \left(\overrightarrow{max} - \overrightarrow{min}\right)_{\mathbf{G}} \ ;$$

– étape 2 : (*cf* B.2.2.2)

$$\left(\overrightarrow{min}\right)_{\mathbf{S}} . \left|Ent(v) \quad 1\right| . \left(\begin{array}{c} \lambda' \\ \overrightarrow{max} - \overrightarrow{min} \end{array}\right)_{\mathbf{G}} \ ;$$

– étape 3 : (*cf* B.2.2.1)

$$\left(\overrightarrow{min}\right)_{\mathbf{S}} . \left|v \times \Lambda' \quad Ent(v) \quad 1\right| . \left(\begin{array}{c} q/\lambda' \\ \lambda' \\ \overrightarrow{max} - \overrightarrow{min} \end{array}\right)_{\mathbf{G}} \ ;$$

– étape 4 : (*cf* B.2.1)

$$\left(\overrightarrow{min}\right)_{\mathbf{S}} . \left|(P_{\mathcal{M}}\, F_{\mathcal{M}}) \times \Lambda' \quad Ent(P_{\mathcal{M}} F_{\mathcal{M}}) \quad 1\right| . \left(\begin{array}{c} \left(\begin{array}{c} \lambda \\ D_{op,2} \end{array}\right)/\lambda' \\ \lambda' \\ \overrightarrow{max} - \overrightarrow{min} \end{array}\right)_{\mathbf{G}} \ ;$$

– étape 5 : nous avons ici l'expression exacte du macro-ajustage opérande nous allons donc renommer les variables suivant le modèle utilisé pour le macro-ajustage résultat,

$$(S')_{\mathbf{S}} . \left|F'_{\mathcal{M}}\right| . (D')_{\mathbf{G}} \ ;$$

– étape 6 : (*cf* B.2.1 page 128)

$$(Q_1)_{\mathbf{M}} . (S')_{\mathbf{S}} . \left|P'_{\mathcal{M}} \quad F'_{\mathcal{M}} \quad 0\right| . \left(\begin{array}{c} Q' \\ D'_{op} \\ \lambda \end{array}\right)_{\mathbf{G}} . \overline{\underline{\Lambda \quad 0 \quad 1}} \ ;$$

– étape 7 : dernière étape, on décide d'agrandir les espaces de départ et d'arrivée en ne passant donc plus de Q_1 à Q_2, mais de $Q_1 D_1$ à $Q_2 D_2$. Ainsi, nous restons cohérents sur les espaces en présence avant le calcul,

$$\left(\begin{array}{c} Q_1 \\ D_{op,1} \\ D_{res,1} \end{array}\right)_{\mathbf{M}} . \left(\begin{array}{c} S' \\ 0 \\ 0 \end{array}\right)_{\mathbf{S}} . \left|\begin{array}{ccccccc} P'_{\mathcal{M}} & F'_{\mathcal{M}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right| . \left(\begin{array}{c} Q' \\ D'_{op} \\ D_{op,1} \\ D_{res,1} \\ \lambda \\ D_{op,2} \\ D_{res,2} \end{array}\right)_{\mathbf{G}} \overline{\underline{\begin{array}{ccccccc} \Lambda & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}}} \ ;$$

On distingue clairement les différentes parties de notre expression : macro-pavage (Q'), macro-ajustage opérande (D'_{op}), macro-ajustage résultat (λ). et enfin l'ajustage spatial opérande ($D_{op,1}$, $D_{res,1}$) et résultat ($D_{op,2}$, $D_{res,2}$). En outre, on remarquera que ces dimensions sont générées complètement.

B.4 Derniers obstacles avant l'obtention d'une tâche ARRAY-OL

Nous appliquons ici les simplifications proposées précédemment (*cf* 2.4.3.2 page 57). En effet, la tâche peut produire des calculs redondants à cause de vecteurs colinéaires issus des diverses multiplication de projections et elle peut contenir des vecteurs inutiles dus notamment à l'utilisation de la segmentation du gabarit.

Nous avons donc réussi à transformer l'expression entre Q_1D_1 et Q_2D_2 sous la forme d'une tâche ARRAY-OL. Nous remplaçons donc l'ancienne expression par la nouvelle.

$$\underbrace{\left(M_1\right)_{\mathbf{M}}.\left(S_1\right)_{\mathbf{S}}.\left|P_{op,1}\quad F_{op,1}\quad 0\right|.\begin{pmatrix}Q_1\\D_{op,1}\\D_{res,1}\end{pmatrix}_{\mathbf{M}}.\begin{pmatrix}S'\\0\\0\end{pmatrix}_{\mathbf{S}}.\left|\begin{array}{ccccccc}P'_{\mathcal{M}}&F'_{\mathcal{M}}&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&0&1&0&0&0\end{array}\right|.\begin{pmatrix}Q'\\D'_{op}\\D_{op,1}\\D_{res,1}\\\lambda\\D_{op,2}\\D_{res,2}\end{pmatrix}_{\mathbf{G}}.\left|\begin{array}{ccccccc}\Lambda&0&0&0&1&0&0\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\end{array}\right|}_{\text{relation } Q_1D_1 \rightarrow Q_2D_2 \text{ ne passant plus par } A_2}.\left|P_{res,2}\quad 0\quad F_{res,2}\right|$$

Nous voulons désormais hierarchiser cette tâche et dans un premier temps calculer la tâche supérieure. Or pour obtenir cette dernière, il nous suffit de manipuler notre nouvelle expression ce qui serait tres simple (grâce à la composition des ODT) si il n'y avait pas le modulo central $\begin{pmatrix}Q_1\\D_{op,1}\\D_{res,1}\end{pmatrix}_{\mathbf{M}}$.

B.4.1 Suppression du modulo.

C'est un des points faibles de la fusion, il n'est pas possible de supprimer automatiquement le modulo, il faut faire des tests avec les valeurs pour supprimer le modulo.

La première technique est d'utiliser le cas général de la partie sur la suppression du modulo (*cf* 2.4.3.3 page 60). La deuxième consiste à vérifier son utilité en calculant les extremums générés par les itérations dans le « super QD ». Si les extremums sont compris entre $\overrightarrow{0}$ et la valeur du modulo, alors on peut effectivement le supprimer.

Mais si aucune de ces techniques ne fonctionne, la fusion n'est malheureusement pas possible.

B.4.2 Problème du décalage.

Plus rien ne s'oppose désormais d'un point de vue strictement mathématique au calcul de la tâche supérieure et des deux sous-tâches. Cependant comme cela a déja été mentionné dans ce rapport, les ODT nécessitent une lecture à plusieurs niveaux et on ne peut se contenter d'une « simple » lecture mathématique. En effet, il est possible, dans certains cas, de générer des résultats incohérents à partir de notre expression.

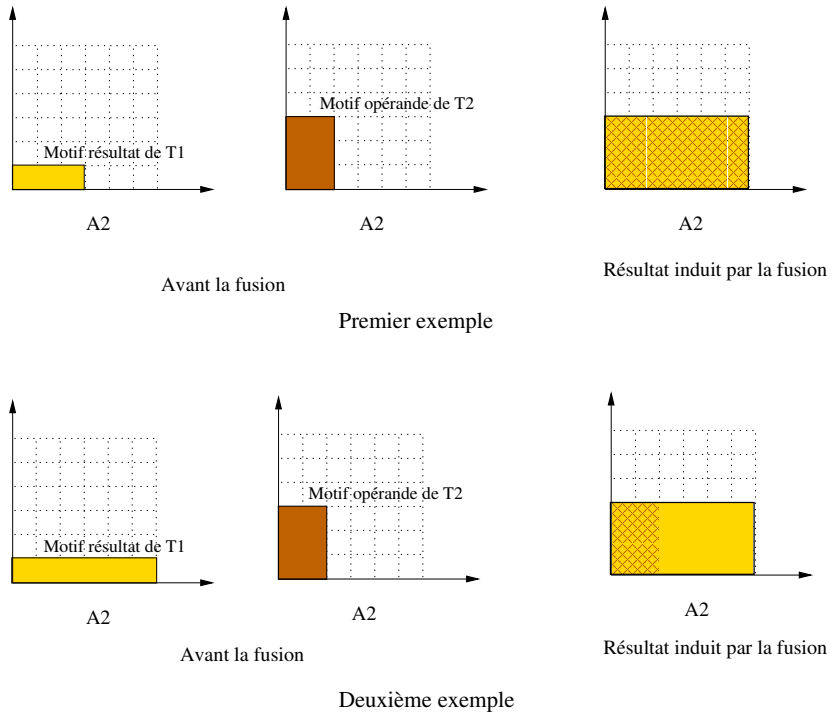


FIG. B.1: Deux exemples pour problème du décalage des origines

Afin de bien cerner le problème, il est nécessaire de revenir sur la construction des macro-motifs et notamment sur la corrélation entre macro-motif opérande et résultat. Tout d'abord considérons le premier exemple de la figure B.1. Dans sa partie gauche, la figure représente les motifs opérandes et résultats du tableau $A2$. C'est ici que se joue la fusion. En effet, si on connaît le nombre de motifs résultats de $T1$ nécessaires pour avoir au moins un motif opérande de $T2$ alors la fusion est possible.

Examinons le processus de fusion pour cet exemple. La matrice de pavage résultat de $T1$ ($P_{res,1}$) vaut $\left(\begin{smallmatrix}0&3\\1&0\end{smallmatrix}\right)$ avec comme bornes (Q_1) $\left(\begin{smallmatrix}\infty\\2\end{smallmatrix}\right)$ et la matrice de pavage opérande de $T2$ ($P_{op,2}$) vaut $\left(\begin{smallmatrix}0&2\\3&0\end{smallmatrix}\right)$ avec comme bornes (Q_2) $\left(\begin{smallmatrix}\infty\\3\end{smallmatrix}\right)$. On cherche une matrice $\mathcal{M}$ telle que $\mathcal{M} \times P_{res,1} = P_{op,2}$. On suit le raisonnement décrit ci-dessus, en calculant $P'_{res,1} \times P_{op,2}$ et on obtient $\left(\begin{smallmatrix}3&0\\0&\frac{2}{3}\end{smallmatrix}\right)$.

Or cette matrice n'a pas ses coefficients dans $\mathbb{Z}$, on utilise donc la segmentation du gabarit pour revenir à une matrice à coefficients entiers ce qui nous donne $\left(\begin{smallmatrix}3&0\\0&2\end{smallmatrix}\right)$ comme macro-pavage opérande ($P'_{\mathcal{M}}$) et $\left(\begin{smallmatrix}1&0\\0&3\end{smallmatrix}\right)$ comme macro-pavage résultat (Λ). Finalement nous obtenons $\left(\begin{smallmatrix}0&6\\3&0\end{smallmatrix}\right)$ comme nouvelle matrice résultat sur $T1$ ($P_{res,1} \times P'_{\mathcal{M}}$) et aussi comme nouvelle matrice de pavage opérande ($P_{op,2} \times \Lambda$) de $T2$.

Le résultat est optimum puisque comme nous le voyons sur la partie droite de la figure nous consommons et nous produisons un motif de même taille sur $A2$.

Regardons maintenant l'example 2, il ne diffère que tres légèrement du premier : la seule différence réside dans le fait que $T1$ produise $A2$ ligne par ligne et non plus demi-ligne par demi-ligne. On est en droit de penser que le problème est le même et que le résultat est identique. Mais il n'en est rien car la matrice de pavage résultat de $T1$ ($P_{res,1}$) ne comporte

qu'un seul vecteur (une seule dimension étant pavée). $P_{res,1}$ vaut $\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$. Or si on effectue le calcul précédent, on obtient $\left(\begin{smallmatrix}0&0\\3&0\end{smallmatrix}\right)$ comme nouvelle matrice de pavage résultat de $T1$, mais $\left(\begin{smallmatrix}0&2\\3&0\end{smallmatrix}\right)$ comme nouvelle matrice de pavage opérande de $T2$. Le résultat n'est plus optimum comme le montre le partie de droite de notre figure, et un recalcul est donc induit.

À ce stade de l'explication, le lecteur peut croire que le seul problème réside dans la présence de ce recalcul, mais ce n'est pas le cas. En effet, il n'est pas possible d'exprimer sous une forme ARRAY-OL correcte le résultat que nous avons obtenu. Notre tâche hiérarchique finale va produire à chaque itération de pavage sur $T3$ un macro-motif résultat de $T1$ et consommera un macro-motif opérande de $T2$. Mais ce dernier est resté de la taille du motif original. Donc, il faudra plusieurs itération de pavage de $T3$ pour produire une ligne complète de $A3$, et à chacune de ces itérations $T2$ consommera une partie différente du motif produit par $T1$ (*cf* figure B.2). Le problème est qu'il n'est pas possible d'exprimer ce décalage en ARRAY-OL car il dépend des itérations de la tâche supérieure et rend donc l'utilisation du shift impossible.

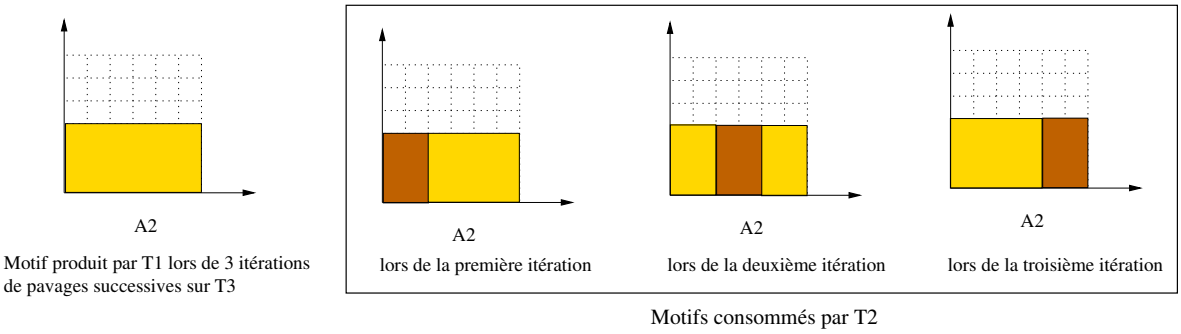


FIG. B.2: Exemple de décalage du macro-motif opérande de $T2$

La solution consiste à calculer puis à comparer les nouvelles matrices de pavage résultats et opérandes sur $T2$. Si il y a des vecteurs différents nous forçons la(es) dimension(s) de macro-pavage incriminée(s) à passer dans le macro-ajustage ce qui résoud en même temps le problème du recalcul. La forme finale de notre expression (« super QD ») reliant Q_1D_1 à Q_2D_2 s'en trouve profondément modifiée puisque tout le macro pavage et tout le macro ajustage est changée. Notre expression devient :

$$\underset{\mathbf{M}}{\begin{pmatrix} Q_1 \\ D_{op,1} \\ D_{res,1} \end{pmatrix}} \underset{\mathbf{S}}{\begin{pmatrix} S' \\ 0 \\ 0 \end{pmatrix}} . \left|\begin{array}{ccccccc} P_{op} & F_{\mathcal{M},op} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right| . \underset{\mathbf{G}}{\begin{pmatrix} Q \\ D_{\mathcal{M},op} \\ D_{op,1} \\ D_{res,1} \\ D_{\mathcal{M},res} \\ D_{op,2} \\ D_{res,2} \end{pmatrix}} . \left|\begin{array}{ccccccc} P_{res} & 0 & 0 & 0 & F_{\mathcal{M},res} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right|$$

B.5 Création de la hiérarchie

On obtient finalement :

$$(M_1)_{\mathbf{M}}.(S_1)_{\mathbf{S}}.\left|P_{op,1}\quad F_{op,1}\quad 0\right|.\underbrace{\begin{pmatrix}S'\\0\\0\end{pmatrix}_{\mathbf{S}}.\left|\begin{matrix}P_{op}&F_{\mathcal{M},op}&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&0&1&0&0&0\end{matrix}\right|.\begin{pmatrix}Q\\D_{\mathcal{M},op}\\D_{op,1}\\D_{res,1}\\D_{\mathcal{M},res}\\D_{op,2}\\D_{res,2}\end{pmatrix}_{\mathbf{G}}.\left|\begin{matrix}P_{res}&0&0&0&F_{\mathcal{M},res}&0&0\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\end{matrix}\right|}_{\text{relation } Q_1D_1\rightarrow Q_2D_2 \text{ ne passant plus par } A_2}.\left|P_{res,2}\quad 0\quad F_{res,2}\right|$$

B.5.1 Calcul de la tâche supérieure.

– On utilise la composition du shift pour obtenir (*cf* 2.4.3.1 page 56) :

$$(M_1)_{\mathbf{M}}.(S_1)_{\mathbf{S}}.(P_{op,1}\times S')_{\mathbf{S}}.\left|P_{op,1}\quad F_{op,1}\quad 0\right|.\left|\begin{matrix}P_{op}&F_{\mathcal{M},op}&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&0&1&0&0&0\end{matrix}\right|.\begin{pmatrix}Q\\D_{\mathcal{M},op}\\D_{op,1}\\D_{res,1}\\D_{\mathcal{M},res}\\D_{op,2}\\D_{res,2}\end{pmatrix}_{\mathbf{G}}.\left|\begin{matrix}P_{res}&0&0&0&F_{\mathcal{M},res}&0&0\\0&0&0&0&0&1&0\\0&0&0&0&0&0&1\end{matrix}\right|.\left|P_{res,2}\quad 0\quad F_{res,2}\right|$$

– On fusionne les deux shifts, les deux projections, les deux segmentations (*cf* 2.4.3.1 page 56) :

$$(M_1)_{\mathbf{M}}.(S_1+P_{op,1}\times S')_{\mathbf{S}}.\left|P_{op,1}\times P_{op}\quad P_{op,1}\times F_{\mathcal{M},op}\quad F_{op,1}\quad 0\quad 0\quad 0\quad 0\right|.\begin{pmatrix}Q\\D_{\mathcal{M},op}\\D_{op,1}\\D_{res,1}\\D_{\mathcal{M},res}\\D_{op,2}\\D_{res,2}\end{pmatrix}_{\mathbf{G}}.\left|\begin{matrix}P_{res,2}\times P_{res}&0&0&0&P_{res,2}\times F_{\mathcal{M},res}&0&F_{res,2}\end{matrix}\right|$$

– On simplifie $D_{res,1}$ et $D_{op,2}$

$$\left(M_1\right)_{\mathbf{M}}.\left(S_1+P_{op,1}\times S'\right)_{\mathbf{S}}.\left|P_{op,1}\times P_{op}\quad P_{op,1}\times F_{\mathcal{M},op}\quad F_{op,1}\quad 0\quad 0\right| .\begin{pmatrix} Q\\ D_{\mathcal{M},op}\\ D_{op,1}\\ D_{\mathcal{M},res}\\ D_{res,2}\end{pmatrix}_{\mathbf{G}}.\overline{P_{res,2}\times P_{res}\quad 0\quad 0\quad P_{res,2}\times F_{\mathcal{M},res}\quad F_{res,2}}$$

B.5.2 Calcul des deux sous-tâches

Nous avons donc calculé la valeur de la tâche supérieure, il faut donc que nous calculions les deux sous-tâches. Ces deux tâches sont équivalentes à $T1$ et $T2$ puisqu'elles doivent effectuer les mêmes traitements, et par conséquent, elles consomment et produisent les mêmes motifs. Or la première tâche de la hierarchie doit consommer le macro-motif opérande de la tâche supérieure. De plus, ce macro-motif est l'agglomeration de plusieurs motifs de la tâche d'origine $T1$, il suffit donc d'utiliser une matrice identité pour paver et ajuster ce macro-motif dans la sous-tâche. Le raisonnement est le même pour montrer que la segmentation de la seconde tâche de la hiérarchie est également la matrice identité. Procédons aux calculs, nous savons que les deux premières doivent être de la forme :

– pour la première sous-tâche :

$$\begin{pmatrix} D_{\mathcal{M},op}\\ D_{op,1}\end{pmatrix}_{\mathbf{M}}.\begin{pmatrix}0\\0\end{pmatrix}_{\mathbf{S}}.\begin{vmatrix}1&0&0\\0&1&0\end{vmatrix}.\begin{pmatrix} D_{\mathcal{M},op}\\ D_{op,1}\\ D_{res,1}\end{pmatrix}_{\mathbf{G}}.\overline{?\quad 0\quad F_{res,1}}.\left(?\right)_{\mathbf{S}}.\left(?\right)_{\star}$$

– pour la deuxième sous-tâche :

$$\left(?\right)_{\mathbf{M}}.\left(?\right)_{\mathbf{S}}.\left|?\quad F_{op,2}\quad 0\right|.\begin{pmatrix} D_{\mathcal{M},res}\\ D_{op,2}\\ D_{res,2}\end{pmatrix}_{\mathbf{G}}.\overline{\begin{matrix}1&0&0\\0&0&1\end{matrix}}$$

Pour obtenir les dernières informations concernant les deux sous-tâches on part d'un point se trouvant dans l'espace d'ajustage opérande ou résultat de notre tâche « super QD ». Ensuite, on parcourt « le chemin » reliant ce point au point correspondant de $A2$. Pour un point se trouvant dans l'ajustage opérande, on passe par la projection et le shift de la « super QD » pour arriver dans Q_1D_1. Puis on passe par la segmentation de $T1$ pour arriver finalement dans $A2$. On compose donc ces ODT pour obtenir la première sous-tâche, et le raisonnement est le même pour la deuxième.

– pour la première sous-tâche :

$$\begin{pmatrix} D_{\mathcal{M},op}\\ D_{op,1}\end{pmatrix}_{\mathbf{M}}.\begin{pmatrix}0\\0\end{pmatrix}_{\mathbf{S}}.\begin{vmatrix}1&0&0\\0&1&0\end{vmatrix}.\begin{pmatrix} D_{\mathcal{M},op}\\ D_{op,1}\\ D_{res,1}\end{pmatrix}_{\mathbf{G}}.\overline{P_{res,1}\times F_{\mathcal{M},op}\quad 0\quad F_{res,1}}.\left(P_{res,1}\times S'\right)_{\mathbf{S}}.\left(M_2\right)_{\star}$$

– pour la deuxième sous-tâche :

$$\left(M_2\right)_{\mathbf{M}}.\left(S_2\right)_{\mathbf{S}}.\left|P_{op,2}\times F_{\mathcal{M},res}\quad F_{op,2}\quad 0\right|.\begin{pmatrix} D_{\mathcal{M},res}\\ D_{op,2}\\ D_{res,2}\end{pmatrix}_{\mathbf{G}}.\overline{\begin{matrix}1&0&0\\0&0&1\end{matrix}}$$

B.5.3 Conclusion

Le calcul d'une fusion est long et complexe. La nécessité d'une lecture à « plusieurs niveaux » n'aide pas à la compréhension. Pourtant cette compréhension est indispensable à la réalisation de l'implémentation.

B.6 Généralisation de la fusion

La généralisation de la fusion telle qu'elle est décrite dans la section 3.2.3 page 71 repose sur les mêmes principes qu'une fusion « simple ». Seul le cas des tableaux intermédiaires liés uniquement à une des deux tâches présente une petite différence. Nous présentons ici les calculs pour ce cas particulier. Soit O_1 un tableau de sortie de la première tâche qui ne soit pas lié à la deuxième et soit I_2 un tableau d'entrée de la deuxième tâche qui ne soit pas lié à la première. La représentation ODT des « demi-tâches » est :

– Pour O_1 :

$$\left(M_{O_1}\right)_{\mathbf{M}} . \left(S_{O_1}\right)_{\mathbf{S}} . \left|P_{O_1} \quad F_{O_1}\right| . \begin{pmatrix} Q_1 \\ D_{O_1} \end{pmatrix}_{\mathbf{G}}$$

– Pour I_2 :

$$\left(M_{I_2}\right)_{\mathbf{M}} . \left(S_{I_2}\right)_{\mathbf{S}} . \left|P_{I_2} \quad F_{I_2}\right| . \begin{pmatrix} Q_2 \\ D_{I_2} \end{pmatrix}_{\mathbf{G}}$$

Le calcul pour la tâche supérieur s'effectue de la manière que pour une tâche simple (*cf* section B.5.1 page 134) :

– Pour O_1 :

$$\left(M_{O_1}\right)_{\mathbf{M}} . \left(S_{O_1}\right)_{\mathbf{S}} . \left|P_{O_1} \times P_{op} \quad F_{O_1} \times F_{\mathcal{M},op} \quad F_{O_1}\right| . \begin{pmatrix} Q \\ D_{\mathcal{M},op} \\ D_{O_1} \end{pmatrix}_{\mathbf{G}}$$

– Pour I_2 :

$$\left(M_{I_2}\right)_{\mathbf{M}} . \left(S_{I_2} + P_{I_2} \times S'\right)_{\mathbf{S}} . \left|P_{I_2} \times P_{res} \quad F_{I_2} \times F_{\mathcal{M},res} \quad F_{I_2}\right| . \begin{pmatrix} Q \\ D_{\mathcal{M},res} \\ D_{I_2} \end{pmatrix}_{\mathbf{G}}$$

Le calcul pour les tâches inférieures s'effectue aussi similairement à celui d'une tâche simple (*cf* section B.5.2) :

– Pour O_1 :

$$\begin{pmatrix} D_{\mathcal{M},op} \\ D_{O_1} \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} . \begin{pmatrix} D_{\mathcal{M},op} \\ D_{O_1} \end{pmatrix}_{\mathbf{G}}$$

– Pour I_2 :

$$\begin{pmatrix} D_{\mathcal{M},res} \\ D_{I_2} \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} 0 \\ 0 \end{pmatrix}_{\mathbf{S}} . \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} . \begin{pmatrix} D_{\mathcal{M},res} \\ D_{I_2} \end{pmatrix}_{\mathbf{G}}$$

Annexe C

Application d'une fusion sur des exemples concrets

FIG. C.1: Légende pour les exemples

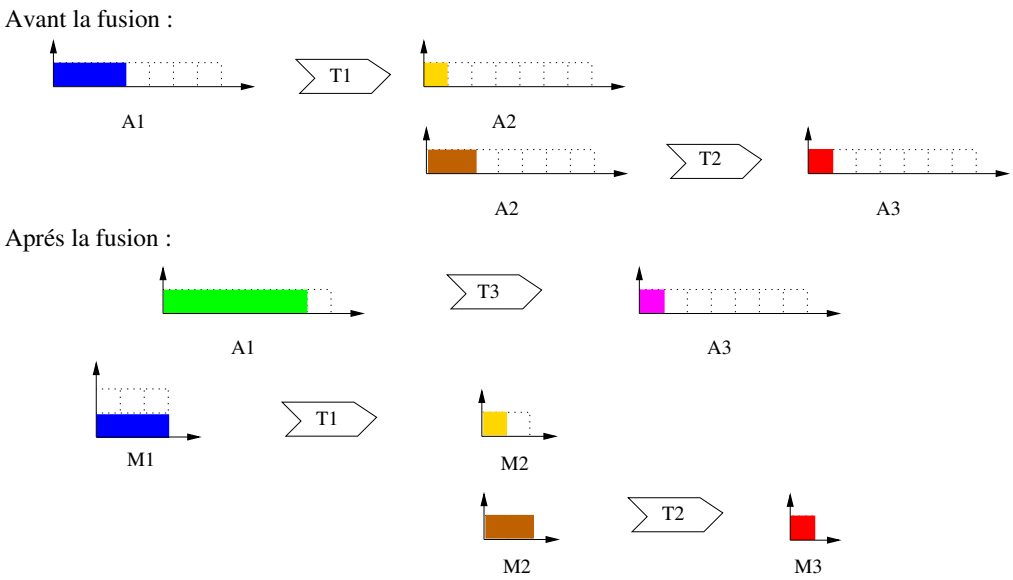


FIG. C.2: Fusion triviale

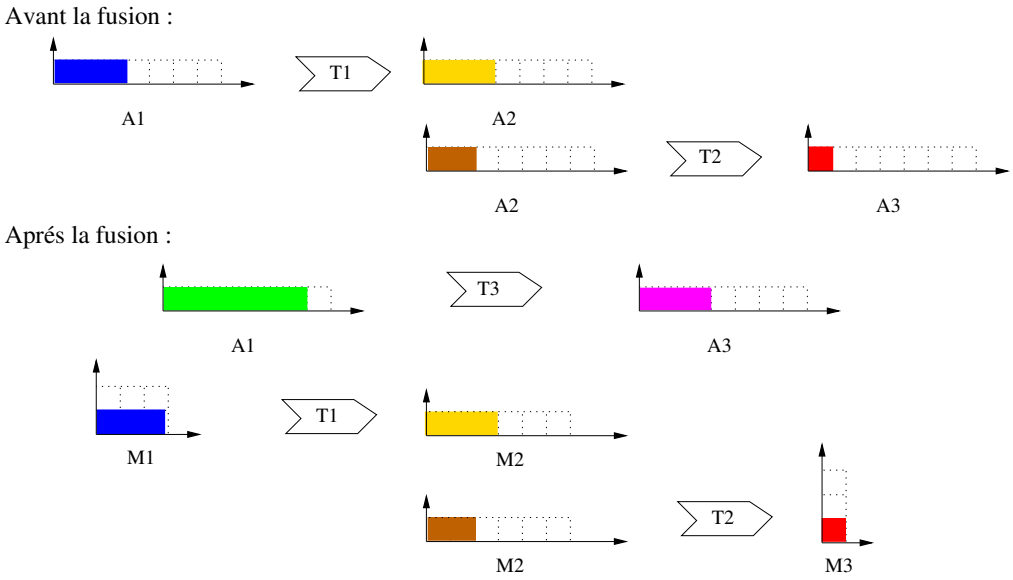


FIG. C.3: Fusion avec création d'un macro-motif résultat : Cas très intéressant car il montre que la fusion produit ici le résultat optimal.

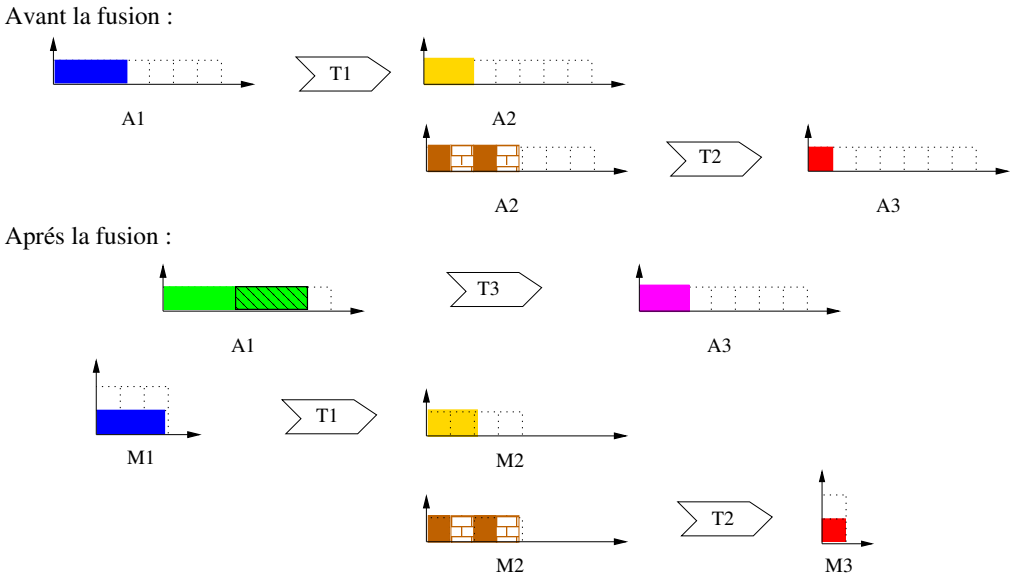


FIG. C.4: Fusion entraînant un recalcul : cas défavorable où la fusion entraine un recalcul à cause d'un recouvrement sur le tableau intermédiaire (*cf* section 3.4.1 page 79).

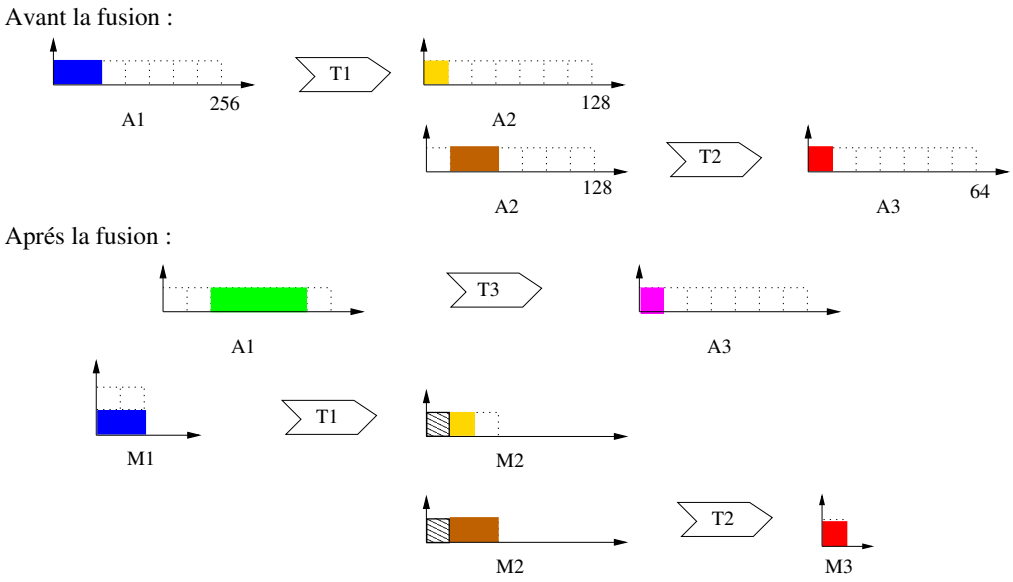


FIG. C.5: Fusion produisant une sous-tâche non exacte : autre cas défavorable (*cf* section 3.3 page 78).

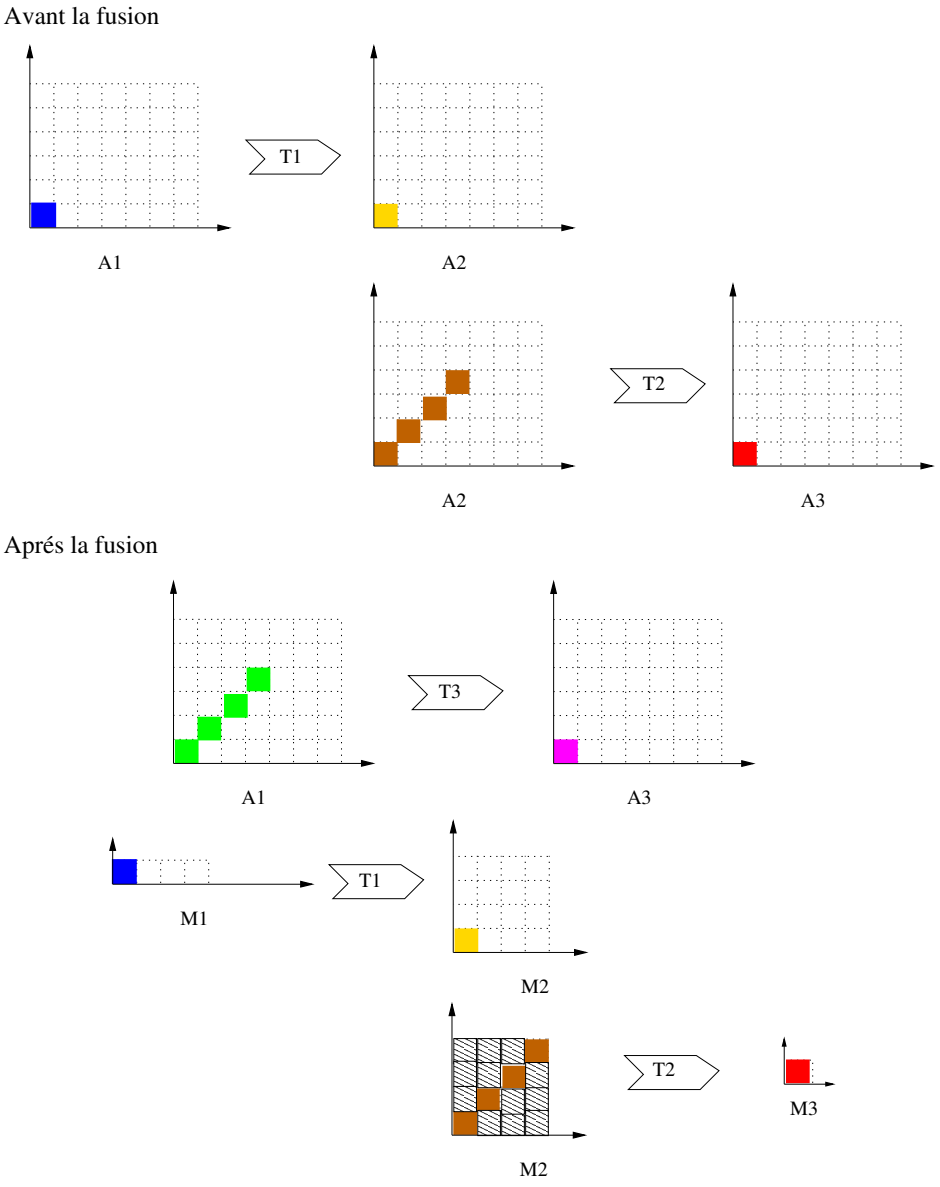


FIG. C.6: Fusion avec une réduction non optimale du tableau intermédiaire : inconvénient mineur de la fusion (*cf* section 3.3 page 78).

Spécification multidimensionnelle pour le traitement du signal systématique

Résumé : De nombreuses applications de traitement du signal ont à traiter des données comportant plusieurs dimensions, pourtant il n'existe que très peu de modèles qui soient capables de tenir compte de cet aspect multidimensionnel. Nous nous sommes donc intéressés à ce problème, mais en nous limitant au traitement du signal systématique (TSS) qui consiste en l'application de traitements très réguliers indépendants des données.

Nous abordons tout d'abord la problématique de la modélisation des applications. Nous comparons différents modèles reposant sur les flux de données synchrones. Puis, nous nous intéressons à ARRAY-OL qui est un modèle de description ayant la faculté d'exprimer les dépendances de données. Mais ce dernier ne fournit aucune méthodologie pour l'exécution des applications et aucun biais d'optimisation. Il faut donc projeter ARRAY-OL sur un modèle de calcul pour exécuter des applications, mais en ayant proposé au préalable une phase d'optimisations.

Nous étudions pour cela différentes méthodes : nous analysons les possibilités offertes par les transformations de boucles, puis nous présentons le formalisme ODT. Nous montrons que les ODT sont en mesure d'exprimer parfaitement les dépendances de données. À l'aide des ODT, nous proposons une suite de transformations constituant une « boîte à outils » capable d'effectuer de simples modifications ou des optimisations plus complexes sur des applications ARRAY-OL. Enfin, nous analysons la projection d'ARRAY-OL vers des modèles de calculs et nous étudions l'impact de nos transformations.

Mots clés : flots de données, transformation de code, compilation, traitement du signal, spécification multidimensionnelle, fusion de boucles

Multidimensional specification for systematic signal processing

Abstract : Many signal processing applications have to manipulate multidimensional data ; however there are only few models of specifications which are able to handle such data. Thus, we study this problem, but we are limiting our approach to systematic signal processing which is characterised by the application of very regular treatments, independent from the data values.

We analyse the modelling of such applications. Firstly, we compare different models based on the synchronous dataflow paradigm and then we introduce ARRAY-OL. ARRAY-OL is a model of description which is able to express data dependences. However, ARRAY-OL does not explain how to execute the applications and does not provide any methodology for optimisations. Therefore, in the case of using ARRAY-OL, it is necessary to suggest some optimisations and to analyse its projections on a computation model.

Hence, we analyse the possibilities offered by the loops transformations and we depict the ODT formalism. We show that the ODT are perfectly suitable for expressing the data dependences. Using it, we recommend several transformations constituting a toolbox which is able to realise simple modifications or more complex optimisations on ARRAY-OL applications. Lastly, we examine the projection of ARRAY-OL on different models of computations.

Keywords : dataflow, code transformation , compilation, signal processing, multidimensional specification, loop fusion