École des mines de Paris
Collège doctoral

N° attribué par la bibliothèque
/__/__/__/__/__/__/__/__/__/__/

Thèse

pour obtenir le grade de

Docteur de l'École des Mines de Paris
Spécialité : Informatique Temps Réel, Robotique et Automatique

présentée et soutenue publiquement
par

Youcef BOUCHEBABA

le 19 novembre 2002

Optimisation des transferts de données pour le traitement du signal : pavage, fusion et réallocation des tableaux

Directeur de thèse : François IRIGOIN

Jury

Madame	Catherine MONGENET	Rapporteur et Président
Monsieur	Jean-Frédéric MYOUPO	Rapporteur
Monsieur	Fabien COELHO	Examinateur
Monsieur	Alain DARTE	Examinateur
Monsieur	François IRIGOIN	Examinateur
Monsieur	Thierry OMNÈS	Examinateur

À mes parents, Baya BOUHA et Ameur BOUCHEBABA

Remerciements

Je tiens tout d'abord à remercier Monsieur le Professeur Robert Mahl, directeur du Centre de recherche en informatique de l'École des mines de Paris de m'avoir permis de faire ma thèse dans son laboratoire.

Je suis très reconnaissant à Fabien Coelho et à François Irigoin d'avoir dirigé mes travaux. Ce travail n'aurait pas été possible sans Fabien Coelho qui m'a proposé un sujet déjà bien défini, mais suffisamment large pour laisser place à la recherche. Il m'a beaucoup encouragé et m'a apporté une aide inestimable tant sur le plan conceptuel que sur le plan pratique. Je tiens aussi à exprimer l'estime que j'ai pour lui pour ses qualités humaines et à le remercier de tout coeur. Le rôle de François Irigoin a été déterminant. Il n'a ménagé ni son temps, ni sa peine, pour guider le choix d'un sujet intéressant, pour lire les rédactions intermédiaires et pour suggérer des solutions. Ses qualités humaines et scientifiques resteront un exemple que j'essaierai de suivre.

Je désire remercier les professeurs Catherine Mongenet et Jean Frédéric Myoupo d'avoir accepté de rapporter ma thèse. Mes remerciements vont aussi à Messieurs Alain Darte et Thierry Omnès pour avoir accepté de faire partie de mon jury. Les commentaires de ce dernier m'ont été d'une grande utilité pour rédiger la version finale de ce mémoire.

J'aimerai également remercier tous les membres de l'équipe PIPS. Les remarques et conseils de Corinne Ancourt et Pierre Jouvelot ont toujours été pertinents. Un grand merci à Georges-André Silber qui m'a aidé sans compter pour effectuer les expériences sur les défauts de cache et qui a su suppléer aux absences de mes encadrants habituels.

Je remercie aussi tous mes amis et en particulier Laure, Pascale, Mahdi, Hakim et Riad avec qui j'ai passé les meilleurs moments à fontainebleau.

Finalement, je remercie mes parents qui m'ont toujours poussé à poursuivre mes études.

Table des matières

Chapitre 1

Introduction

Avec la multiplication des systèmes embarqués, la réduction de la consommation de l'énergie électrique est devenue un problème crucial pour l'industrie comme le montre *l'International Technology Roadmap for Semiconductors (http ://public.itrs.net)* où il est classé comme un des deux défis technologiques les plus importants. Ce problème attire aussi de nombreux chercheurs dans la communauté de la compilation et des transformations de programmes [73, 71, 61, 20, 86]. D'après de nombreuses études [42, 44, 19, 22, 33], cette consommation dépend fortement des échanges de signaux entre circuits et donc des transferts de données entre les différents niveaux de la hiérarchie mémoire : les registres, les caches et la mémoire centrale. Dans la littérature, la réduction de ces transferts est connue sous le nom du problème de l'augmentation de la localité des données. La réduction de ces transferts permet d'obtenir des gains significatifs en termes d'économie de l'énergie électrique dans les systèmes embarqués.

La recherche dans le domaine des transformations de programmes, domaine important de la compilation, a attiré l'attention de nombreux chercheurs depuis plusieurs années. Elle consiste à trouver de nouvelles techniques permettant au compilateur de transformer les programmes sources en d'autres programmes qui optimisent le mieux certains critères, comme par exemple le parallélisme, la vitesse d'exécution ou encore la localité des données qui a entre autres pour effet de réduire la consommation d'énergie. Dans ce domaine particulier, les chercheurs de l'IMEC ont proposé diverses techniques et méthodologies ayant pour but de réduire le volume mémoire nécessaire [34, 35, 36], de modifier le flot de contrôle [72], de transformer les boucles [45, 46] et de généraliser le problème au cas des systèmes ayant des hiérarchies de mémoire [81, 82].

Lors de l'optimisation des programmes, le plus grand gain vient de l'optimisation des nids de boucles, car ils monopolisent la plupart du temps de calcul des programmes scientifiques. C'est à eux que nous nous intéressons dans cette thèse. Par ailleurs, nous avons pour objectif de développer des techniques automatiques, utilisables dans un compilateur, et non des méthodologies de transformations.

Depuis de nombreuses années, plusieurs techniques ont été proposées dans le domaine des transformations de programmes pour transformer ces nids. Parmi ces techniques, nous pouvons citer les transformations unimodulaires et le pavage. Comme toute transformation de programmes, ces deux techniques transforment un nid de boucles en un autre qui possède le même nombre d'itérations mais un ordre d'exécution différent. Les transformations unimodulaires sont des transformations que nous pouvons représenter par des matrices unimodulaires. Une matrice unimodulaire possède trois propriétés importantes : (1) elle est carrée ; elle transforme donc un espace d'itération de dimension n en un autre espace de dimension n, (2) ses éléments sont des entiers ; elle transforme donc un vecteur entier en un autre vecteur entier et (3) la valeur absolue de son déterminant est égale à 1 ; l'espace initial et l'espace transformé ont donc le même volume (les boucles gardent les mêmes pas).

Le pavage n'est pas une transformation unimodulaire. En général, il transforme un nid de boucles de profondeur n en un autre nid de profondeur $2n$. Il est l'une des techniques les plus importantes qui a été introduite initialement par F. Irigoin [39] et M. Wolfe [75, 78] et reprise ensuite par plusieurs auteurs. Cette technique a plusieurs intérêts : (1) maximiser le parallélisme, (2) réduire les communications et (3) optimiser l'utilisation de la hiérarchie mémoire, notamment les caches et les registres. Malheureusement, l'application de ces deux techniques est limitée à des codes particuliers qui sont constitués d'un seul nid de boucles. Dans cette thèse, nous proposons l'extension du pavage à des séquences de nids de boucles et des transformations unimodulaires et des nids de boucles mal imbriquées.

Le modèle polyèdrique [68] est très utilisé dans le domaine des transformations de programmes. Ce modèle nécessite une bonne connaissance d'algèbre linéaire. Dans le *chapitre 2*, nous donnons une définition formelle de la notion de nid de boucles et nous donnons quelques préliminaires d'algèbre linéaire qui sont utilisés tout au long de cette thèse.

Dans les *chapitres* 3, 4 et 5, nous nous sommes intéressés à la réduction de la consommation de l'énergie électrique dans le cas des applications de traitement du signal [59, 37, 60]. Les codes considérés sont des séquences de nids de boucles. Chacun de ces nids utilise un patron de données produites dans le nid précédent. Lors de la conception de ces applications, les programmeurs ne tiennent pas forcément compte du problème de la localité des données. Pour cela, nous sommes amenés à transformer ces programmes en procédant à (1) une fusion avec alignement, (2) à un pavage avec fusion, et (3) à une réallocation de chaque tableau intermédiaire par un ensemble de tampons.

Dans le *chapitre 3*, nous traitons le problème de la fusion d'une séquence de nids de boucles parallèles. Pour rendre cette fusion possible, nous translatons le domaine d'itération de chaque *nid k* avec un délai $\vec{h}_k$. Cela revient à modifier les dépendances entre ces différents nids et à les rendre lexicographiquement positives. Pour augmenter davantage la localité des données, nous remplaçons les tableaux intermédiaires par des tampons circulaires de tailles inférieures. Ces tampons ne contiennent que les données vivantes. Cette étape permet de ne maintenir en mémoire que les données utiles et d'éviter de charger plusieurs fois les mêmes données.

Dans le *chapitre 4*, nous présentons une méthode qui permet d'appliquer simultanément le pavage et la fusion sur une séquence de nids de boucles. Dans ce chapitre nous nous sommes intéressés uniquement aux données vivantes au niveau cache. Notre pavage est donc à un seul niveau ; nous transformons les différents nids qui sont de profondeur n en un seul nid de profondeur $2n$. Pour considérer tous les pavages possibles (les ordonnancements des tuiles et des itérations à l'intérieur de ces tuiles), nous avons représenté notre pavage avec deux matrices : (1) une matrice $\mathcal{A}$ qui permet de spécifier la taille des tuiles et (2) une matrice de permutation $\mathcal{P}$ qui permet de spécifier l'ordonnancement des itérations. Nous avons proposé deux méthodes d'allocation des tampons. Le principe de la première est similaire à celle de la fusion. Par contre dans la deuxième nous remplaçons chaque tableau par un ensemble de sous-tampons.

Le *chapitre 5* est similaire au *chapitre 4*, mais notre pavage est à deux niveaux, c'est-à-dire que nous transformons les différents nids qui sont de profondeur n en un seul nid de profondeur $3n$, car nous nous sommes intéressés aux données vivantes aux niveaux des caches et des registres. Notre pavage est toujours représenté par deux matrices $\mathcal{A}$ et $\mathcal{P}$ mais leurs tailles sont supérieures à celles du *chapitre 4*. Après l'application simultanée du pavage et de la fusion sur le code en entrée, nous déroulons les boucles les plus internes qui parcourent les itérations dans les petites tuiles afin d'éliminer les accès redondants et d'exploiter l'allocation des registres. Dans ce chapitre, les tableaux intermédiaires sont remplacés par un ensemble de registres pour les petites tuiles et de sous-tampons pour les grandes tuiles.

Les optimisations de code à base de transformations unimodulaires ne peuvent s'appliquer qu'à des nids de boucles parfaitement imbriquées alors que les codes réels comportent souvent des nids de boucles mal imbriquées. Enfin, les techniques de pipeline logiciel ne peuvent s'appliquer efficacement qu'à des corps de boucles sans tests. Pour permettre l'application de transformations unimodulaires et de pipeline logiciel à des codes réels, nous proposons dans le *chapitre 6*, une technique d'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées et non paramétriques à deux dimensions. Nous pouvons diviser ce chapitre en deux sous-parties. Dans la première, nous nous intéressons à la transformation d'un nid de boucles mal imbriquées en un nid parfaitement imbriqué et dans la deuxième, nous nous intéressons au problème inverse ; l'élimination des gardes dans le code obtenu après l'application d'une transformation unimodulaire. Dans la première sous-partie, nous transférons toutes les instructions dans la boucle la plus interne et nous ajoutons des gardes nécessaires sur ces dernières et nous donnons la condition nécessaire et suffisante à l'application de cette transformation. Dans la deuxième sous-partie, nous décomposons le nid transformé en une séquence de nids de boucles de manière à toujours avoir une seule contrainte inférieure et une seule contrainte supérieure dans la boucle la plus interne afin de pouvoir sortir les instructions gardées dans cette boucle.

Comme nous l'avons signalé précédemment, l'objectif de ces travaux est d'améliorer la localité des données afin de réduire la consommation de l'énergie électrique. Il n'existe pas réellement de méthodes théoriques parfaites qui permettent de mesurer ces améliorations. Car il y a plusieurs paramètres qui rentrent en jeux : (1) le code lui même, (2) la politique de gestion des différentes mémoires...etc. La méthode la plus simple est de comparer directement les résultats du code original et du code transformé. Pour cela, nous avons implanté dans *PIPS* [64], le compilateur développé à l'École des mines de Paris, toutes les transformations décrites dans cette thèse. Dans le *chapitre 7*, nous décrivons ces implantations et nous présentons les différentes expérimentations qui ont été effectuées afin de mesurer l'efficacité de ces techniques.

Dans le dernier chapitre, nous donnons quelques conclusions et énumérons les améliorations que nous pensons pouvoir apporter à nos différentes techniques dans des travaux futurs.

Chapitre 2

Préliminaires

La notion de nid de boucles est utilisée dans la plupart des travaux sur les transformations de programmes. Dans ce chapitre, nous la définissons formellement dans le cas général et ensuite nous donnons les restrictions que nous avons posées sur les nids utilisés dans cette thèse. Nous rappelons quelques préliminaires et opérateurs en algèbre linéaire que nous avons utilisés pour formaliser nos différentes transformations et prouver les théorèmes, les propositions et les lemmes dans les différents chapitres de cette thèse. La plupart des opérateurs manipulés dans cette thèse sont des opérateurs standards et ils sont utilisés dans les différents travaux existants sur les transformations de programmes et en compilation. Ici, nous nous contentons de rappeler les opérateurs les moins habituels.

2.1 Nid de boucles

Les structures itératives telles que le *do* de Fortran et le *for* de C sont les structures qui ont concentré le plus l'attention des chercheurs dans le domaine des transformations de programmes, à cause du gain qui peut être obtenu lors de leur optimisation. La notion de nid de boucles qui est une suite imbriquée de structures itératives, sera utilisée tout au long de cette thèse ; nous la définissons formellement dans la *définition 1*.

Définition 1 (Nid de boucles) *Nous appelons un nid de boucles une suite imbriquée de structures itératives qui a la forme suivante :*

$$
\begin{array}{l}
\mathsf{do}\ i_1 = O_1,\ N_1,\ P_1 \\
\quad . \\
\quad \mathsf{do}\ i_j = O_j,\ N_j,\ P_j \\
\qquad . \\
\qquad \mathsf{do}\ i_n = O_n,\ N_n,\ P_n \\
\qquad\quad \mathsf{Corps} \\
\qquad \mathsf{enddo} \\
\qquad . \\
\quad \mathsf{enddo} \\
\quad . \\
\mathsf{enddo}
\end{array}
$$

où Corps *est un ensemble d'instructions,* i_j *une variable entière appelée l'indice de la* $j^{\text{ème}}$ *boucle,* O_j *et* N_j *des expressions entières définissant les bornes de la* $j^{\text{ème}}$ *boucle,* P_j *une expression entière dont la valeur est différente de zéro définissant le pas de la* $j^{\text{ème}}$ *boucle.* n *est appelé la profondeur du nid et le vecteur* $\vec{i} = (i_1, \cdots, i_n)^t$ *est appelé le vecteur d'itération.*

Cette définition est très générale. Dans la suite de cette thèse, nous faisons les restrictions suivantes sur les différents nids qui seront manipulés :

1. les expressions O_j et N_j sont des constantes numériques ;
2. les pas P_j sont des constantes égales à 1.

Si $D = \{\vec{i} \in \mathbb{Z}^n \ / \ \forall j \ : \ O_j \leq i_j \leq N_j\}$, le nid donné dans la *définition 1* sera noté dans la suite de cette thèse par :

$$\text{do } \vec{i} \in D$$

$$\text{Corps}$$

$$\text{enddo}$$

Ces restrictions ne sont pas gênantes parce que :

1. on peut toujours réduire le pas à 1 ;
2. les applications embarquées de traitement du signal traitent des objets de tailles fixes.

2.2 Opérateurs arithmétiques

La plupart des opérateurs arithmétiques utilisés dans cette thèse sont des opérateurs classiques et standards. Ici, nous rappelons les trois opérateurs les moins habituels. Ces trois opérateurs sont $\lfloor\ \rfloor$, $\lceil\ \rceil$ et mod qui sont définis comme suit :

Définition 2 ($\lfloor\ \rfloor$ et $\lceil\ \rceil$) *Soit* $x \in R$*, nous notons par :*

1. $\lfloor x \rfloor$*, le plus grand entier non supérieur à* x *;*
2. $\lceil x \rceil$*, le plus petit entier non inférieur à* x*.*

Définition 3 (Modulo) *Soient* a *et* b *deux entiers tel que* $b > 0$ *(*$a \in \mathbb{Z}$ *et* $b \in \mathbb{Z}^+$*), l'opérateur* mod *est défini comme suit* $a \bmod b = a - b\lfloor \frac{a}{b} \rfloor$*. Par conséquent,* $a \bmod b \geq 0$*.*

Dans cette thèse, le cas $b < 0$ n'est pas utilisé.

2.3 Opérateurs sur les vecteurs

Les opérateurs sur les vecteurs sont divisés en deux grandes classes : (1) les opérateurs qui sont *component-wise* et qui sont appliqués indépendamment entre les composantes des vecteurs et (2) les opérateurs qui définissent l'ordre d'exécution des itérations dans les nids de boucles, qui est un ordre lexicographique.

2.3.1 Opérateurs component-wise

Tout au long de cette thèse, le vecteur $(x_1, \cdots, x_n)^t$ sera noté par $\vec{x}$ et les vecteurs $\vec{0}$ et $\vec{1}$ représentent respectivement le vecteur nul qui est égal à $(0, \cdots, 0)^t$ et le vecteur unité qui est égal à $(1, \cdots, 1)^t$. Les opérateurs relationnels $<$, $\leq$, $=$, $>$ et $\geq$ entre deux vecteurs sont component-wise, c'est-à-dire qu'ils sont appliqués entre les composantes des vecteurs.

Définition 4 ($<$, $\leq$, $=$, $>$ et $\geq$) *Soient $\vec{x} = (x_1, \cdots, x_n)^t$ et $\vec{y} = (y_1, \cdots, y_n)^t$. Les opérateurs relationnels $<$, $\leq$, $=$, $>$ et $\geq$ sont définis comme suit :*

1. $\vec{x} < \vec{y}$ *si et seulement si* $\forall k \in [1, \cdots, n] \ : \ x_k < y_k$;
2. $\vec{x} \leq \vec{y}$ *si et seulement si* $\forall k \in [1, \cdots, n] \ : \ x_k \leq y_k$;
3. $\vec{x} = \vec{y}$ *si et seulement si* $\forall k \in [1, \cdots, n] \ : \ x_k = y_k$;
4. $\vec{x} > \vec{y}$ *si et seulement si* $\forall k \in [1, \cdots, n] \ : \ x_k > y_k$;
5. $\vec{x} \geq \vec{y}$ *si et seulement si* $\forall k \in [1, \cdots, n] \ : \ x_k \geq y_k$.

Ce sont des ordres partiels

Définition 5 ($\min_<(D)$ et $\max_<(D)$) *Si $D = \{\vec{i} \in \mathbb{Z}^n\}$, nous notons par $\min_<(D)$ le vecteur $\vec{x}$ telles que $x_j = \min(i_j)$ pour tout $\vec{i} \in D$ et par $\max_<(D)$ le vecteur $\vec{x}$ telles que $x_j = \max(i_j)$ pour tout $\vec{i} \in D$. Les vecteurs $\min_<(D)$ et $\max_<(D)$ ne sont pas nécessairement dans le domaine D.*

2.3.2 Opérateurs lexicographiques

L'ordre total d'exécution des itérations dans un nid de boucles est spécifié par la relation d'ordre lexicographique. Les quatre opérateurs lexicographiques utilisés dans le domaine des transformations de programmes sont $\prec$, $\preceq$, $\succ$ et $\succeq$.

Définition 6 ($\prec$, $\preceq$, $\succ$ et $\succeq$) *Soient $\vec{x} = (x_1, \cdots, x_n)^t$ et $\vec{y} = (y_1, \cdots, y_n)^t$. Nous disons que :*

1. *le vecteur $\vec{x}$ est lexicographiquement inférieur à $\vec{y}$, qui est noté par $\vec{x} \prec \vec{y}$, si et seulement si $(x_1 < y_1)$ ou $(x_1 = y_1$ et $(x_2, \cdots, x_n)^t \prec (y_2 \cdots, y_n)^t)$;*
2. *le vecteur $\vec{x}$ est lexicographiquement inférieur ou égal à $\vec{y}$, qui est noté par $\vec{x} \preceq \vec{y}$ si et seulement si $\vec{x} = \vec{y}$ ou $\vec{x} \prec \vec{y}$;*
3. *le vecteur $\vec{x}$ est lexicographiquement supérieur à $\vec{y}$, qui est noté par $\vec{x} \succ \vec{y}$ si et seulement si $\vec{y} \prec \vec{x}$;*
4. *le vecteur $\vec{x}$ est lexicographiquement supérieur ou égal à $\vec{y}$, qui est noté par $\vec{x} \succeq \vec{y}$ si et seulement si $\vec{y} \preceq \vec{x}$.*

2.3.3 Extensions de certains opérateurs component-wise à des vecteurs

Dans les définitions suivantes, nous procédons à une extension des opérateurs $*$, $\lfloor \ \rfloor$, $\lceil \ \rceil$, mod et / sur les vecteurs. En reprenant les notations de Xue [84], l'extension de la multiplication $*$ est dénotée par o.

Définition 7 **(o)** *Soient* $\vec{x} = (x_1, \cdots, x_n)^t$ *et* $\vec{y} = (y_1, \cdots, y_n)^t$*, nous définissons :*

$$\vec{x}\ o\ \vec{y} = (x_1 y_1, \cdots, x_n y_n)^t$$

Définition 8 **($\lfloor \vec{x} \rfloor$ et $\lceil \vec{x} \rceil$)** *Soit* $\vec{x} = (x_1, \cdots, x_n)^t$*, les opérateurs* $\lfloor\ \rfloor$ *et* $\lceil\ \rceil$ *sont définis sur les vecteurs de la manière suivante :*

$$\lfloor \vec{x} \rfloor = (\lfloor x_1 \rfloor, \cdots, \lfloor x_n \rfloor)^t$$

$$\lceil \vec{x} \rceil = (\lceil x_1 \rceil, \cdots, \lceil x_n \rceil)^t$$

Définition 9 **(/ et** mod**)** *Les opérateurs* / *et* mod *sont définis uniquement sur les vecteurs entiers. Si* $\vec{a} = (a_1, \cdots, a_n)^t$ *et* $\vec{b} = (b_1, \cdots, b_n)^t$ *sont des vecteurs entiers, nous définissons :*

$$\begin{aligned} \vec{a} \bmod \vec{b} &= \vec{a} - \vec{b}\ o\ \lfloor \frac{\vec{a}}{\vec{b}} \rfloor = (a_1 \bmod b_1, \cdots, a_n \bmod b_n)^t \\ \frac{\vec{a}}{\vec{b}} &= (\frac{a_1}{b_1}, \cdots, \frac{a_n}{b_n})^t \end{aligned}$$

Chapitre 3

Fusion d'une séquence de nids de boucles

La fusion de boucles [38, 49, 65, 87, 56, 55] est une technique de transformation de programmes qui permet de combiner plusieurs boucles en une seule. Cette technique présente plusieurs avantages, parmi lesquels nous pouvons citer :

1. la réduction des tests des bornes de boucles [23] (surcoût de contrôle),
2. la réduction de la synchronisation quand les boucles sont distribuées sur plusieurs unités de calcul [14],
3. l'augmentation de la localité des données [57].

Dans ces travaux, nous nous intéressons à ce dernier point qui revient à minimiser les transferts des données entre les différents niveaux de la hiérarchie mémoire, la mémoire centrale, les caches et les registres. Ces transferts jouent un rôle très important dans la minimisation de l'énergie électrique [19].

Dans le cas général, la fusion de deux boucles consécutives est valide si et seulement si elles ont le même domaine d'itération et si dans la boucle fusionné, nous ne créons pas, de l'instruction de la deuxième boucle vers l'instruction de la première boucle [80, 17], une *dépendance de flot*, une *anti-dépendance* ou une *dépendance de sortie* . Par exemple, l'application de la fusion pour les boucles de la *figure 3.1(a)* donne la boucle en *figure 3.1(c).* La seule dépendance qui existe est la *dépendance de flot* de S_1 vers S_2. Par conséquent, la fusion est légale. La fusion des boucles de la *figure 3.1(b)* est donnée en *figure 3.1(d)* : cette fusion n'est pas valide car il existe une *anti-dépendance* de S_2 vers S_1.

Dans ce chapitre, nous montrons comment nous pouvons appliquer la fusion sur une séquence de nids de boucles qui n'ont pas nécessairement le même domaine d'itération mais qui vérifient d'autres hypothèses qui sont données en *section 3.1.* À chaque itération d'un *nid k*, nous calculons un nouvel élément du tableau A_k en fonction d'un patron de données du tableau A_{k-1} qui sont produites dans le *nid k-1.* La seule condition que nous devons garantir pour que notre fusion soit valide consiste à assurer que toutes les données du tableau A_{k-1} nécessaires pour le calcul d'un élément du tableau A_k à une itération $\vec{i}$ dans le nid fusionné ont déjà été calculées par les itérations précédentes. Pour satisfaire cette condition, nous translatons le domaine d'itération D_k de chaque nid avec un délai $\vec{h}_k$.

Dans la pratique, nous avons besoin uniquement du tableau du dernier nid ; les tableaux associés aux autres nids sont des intermédiaires. Par conséquent, nous pouvons remplacer ces tableaux intermédiaires par des tampons de taille inférieure et qui ne contiendront (presque) que les données vivantes. La notion de donnée vivante a été étudiée en détail par Gannon [31] et Eisenbeis [27]. Le maintien des données vivantes en mémoire permet d'éviter de charger

```
do i = 1, n                              do i = 1, n
   a(i) = i         (* s1 *)                a(i) = i            (* s1 *)
enddo                                    enddo
do i = 1, n                              do i = 1, n
   b(i) = a(i)      (* s2 *)                b(i) = a(i+1)       (* s2 *)
enddo                                    enddo
(a)                                      (b)

do i = 1, n                              do i = 1, n
   a(i) = i         (* s1 *)                a(i) = i            (* s1 *)
   b(i) = a(i)      (* s2 *)                b(i) = a(i+1)       (* s2 *)
enddo                                    enddo
(c)                                      (d)
```

Figure 3.1: La fusion des boucles de (a) donnée en (c) est valide. La fusion des boucles de (b) donnée en (d) n'est pas valide à cause de l'*anti-dépendance* de S_2 vers S_1

plusieurs fois la même donnée et par conséquent de réduire les transferts de données entre les différents niveaux de la hiérarchie mémoire.

3.1 Hypothèses sur le code en entrée

Les codes considérés dans ce chapitre et dans les chapitres qui suivent sont des séquences de nids de boucles. Ces codes ont la forme du code donné en *figure 3.2* où $\otimes$ représente une opération quelconque. Tous les nids ont la même profondeur mais elle est quelconque. Les tableaux associés à ces différents nids ont tous la même dimension qui est égale à la profondeur des nids.

do $\vec{i}_1 \in D_1$
  $A_1(\vec{i}_1) = A_0(\vec{i}_1 + \vec{v}_1^1) \otimes .... \otimes A_0(\vec{i}_1 + \vec{v}_{m_1}^1)$ (* s_1 *)
enddo
.
.
do $\vec{i}_k \in D_k$
  $A_k(\vec{i}_k) = A_{k-1}(\vec{i}_k + \vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i}_k + \vec{v}_{m_k}^k)$ (* s_k *)
enddo
.
.
do $\vec{i}_p \in D_p$
  $A_p(\vec{i}_p) = A_{p-1}(\vec{i}_p + \vec{v}_1^p) \otimes .... \otimes A_{p-1}(\vec{i}_p + \vec{v}_{m_p}^p)$ (* s_p *)
enddo

Figure 3.2: Code en entrée (séquence de nids de boucles). $\forall j : \vec{i}_j \in \mathbb{Z}^n$, la dimension des tableaux A_k est n et $\otimes$ représente une opération quelconque qui dépend de k

Chacun de ces nids accède à des données suivant un *patron*, c'est-à-dire des boucles dans lesquelles les références à un même tableau sont égales, à une translation près :

$$\begin{array}{rcl} \vec{i}_k \text{ et } \vec{i}_{k'} \text{ deux références au tableau } A_k & \Rightarrow & \vec{i}_k = \vec{i}_{k'} + \vec{d} \\ & \Rightarrow & \text{les dépendances sont uniformes.} \end{array}$$

Le *patron* de références au tableau A_{k-1} associé au *nid k* est représenté par $R^k = \{\vec{v}_1^k,\ \vec{v}_2^k, \ldots.., \vec{v}_{m_k}^k\}$ et nous supposons que cet ensemble est ordonné lexicographiquement de la manière suivante : $\vec{v}_1^k \preceq \vec{v}_2^k \preceq \ldots.. \preceq \vec{v}_{m_k}^k$.

Nous imposons que ces codes forment des chaînes de tâches (figure *3.3*), c'est-à-dire que chaque nid utilise des données qui sont produites uniquement dans le nid qui le précède (le *nid k* utilise les données qui sont produites dans le *nid k-1*). La généralisation de cette méthode à d'autres classes de codes, par exemple les codes qui sont sous forme de graphe acyclique orienté, nécessite une étude plus détaillée, car le problème de la fusion avec alignement dans le cas général est *NP_complet* [23].

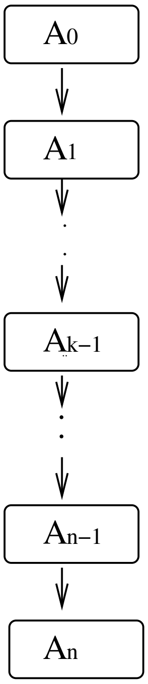


Figure 3.3: Chaîne de tâches

Contraintes sur les domaines associés aux différents nids :

Le seul domaine qui est imposé par l'utilisateur est le domaine D_0 qui est le domaine de définition du tableau A_0 et il est donné par :

$$D_0 = \{\vec{i} :\ \vec{\inf}_0 \leq \vec{i} \leq \vec{\sup}_0\}$$

À chaque nouvelle itération $\vec{i}_k \in D_k$ nous calculons un nouvel élément du tableau A_k en fonction de m_k éléments du tableau A_{k-1}. Pour éviter d'utiliser des éléments du tableau A_{k-1} qui ne sont pas définis dans le domaine D_{k-1}, nous supposons que les domaines $D_{k=1\ldots p}$ sont dérivés de la manière suivante :

$$D_k = \{\vec{i} \mid \forall \vec{v} \in R^k :\ \vec{i} + \vec{v} \in D_{k-1}\} \tag{3.1}$$

À une itération $\vec{i}$ pour calculer $A_k(\vec{i})$ nous utilisons $A_{k-1}(\vec{i}+\vec{v})$ avec $\vec{v} \in R^k$. La *condition 3.1* signifie : pour que $A_{k-1}(\vec{i}+\vec{v})$ soit défini il suffit que $\vec{i}+\vec{v} \in D_{k-1}$.

Proposition 1 *Les bornes des domaines D_k sont des constantes numériques, c'est-à-dire que : $D_k = \{\vec{i} : \ \vec{inf}_k \leq \vec{i} \leq \vec{sup}_k\}$ avec $\vec{inf}_k$ et $\vec{sup}_k \in \mathbb{Z}^n$ et nous avons les relations suivantes pour tout $k \geq 0$:*

$$\vec{inf}_{k+1} = \vec{inf}_k - \vec{L}^{k+1} \text{ avec } L_j^{k+1} = \min_{(\vec{v} \in R^{k+1})}(v_j)$$
$$\vec{sup}_{k+1} = \vec{sup}_k - \vec{U}^{k+1} \text{ avec } U_j^{k+1} = \max_{(\vec{v} \in R^{k+1})}(v_j)$$

Preuve : Nous notons par $P(k)$ la propriété : les bornes du domaine D_k sont des constantes numériques et nous montrons par récurrence que $\forall k : \ P(k)$ est vraie.

1. par définition $D_0 = \{\vec{i} : \ \vec{\mathrm{inf}}_0 \leq \vec{i} \leq \vec{\mathrm{sup}}_0\}$. Par conséquent P(0) est vraie.
2. nous supposons que $P(k)$ est vraie ($D_k = \{\vec{i} : \ \vec{\mathrm{inf}}_k \leq \vec{i} \leq \vec{\mathrm{sup}}_k\}$) et nous montrons par récurrence que $P(k+1)$ l'est aussi.

$$(3.1) \quad \Rightarrow \quad D_{k+1} = \{\vec{i} \mid \forall \vec{v} \in R^{k+1} : \ \vec{i}+\vec{v} \in D_k\}$$

Comme $D_k = \{\vec{i} : \ \vec{\mathrm{inf}}_k \leq \vec{i} \leq \vec{\mathrm{sup}}_k\}$ alors

$$\begin{aligned} D_{k+1} &= \{\vec{i} \mid \forall \vec{v} \in R^{k+1} : \ \vec{\mathrm{inf}}_k \leq \vec{i}+\vec{v} \leq \vec{\mathrm{sup}}_k\} \\ D_{k+1} &= \{\vec{i} \mid \forall \vec{v} \in R^{k+1} : \ \vec{\mathrm{inf}}_k - \vec{v} \leq \vec{i} \leq \vec{\mathrm{sup}}_k - \vec{v}\} \\ D_{k+1} &= \{\vec{i} \mid \forall \vec{v} \in R^{k+1}, \forall j : \ \vec{\mathrm{inf}}_{k,j} - v_j \leq i_j \leq \vec{\mathrm{sup}}_{k,j} - v_j\} \end{aligned}$$

Donc les bornes du domaine D_{k+1} sont des constantes numériques, c'est-à-dire que $D_{k+1} = \{\vec{i} : \ \vec{\mathrm{inf}}_{k+1} \leq \vec{i} \leq \vec{\mathrm{sup}}_{k+1}\}$ tels que :

$$\vec{\mathrm{inf}}_{k+1} = \vec{\mathrm{inf}}_k - \vec{L}^{k+1} \text{ avec } L_j^{k+1} = \min_{(\vec{v} \in R^{k+1})}(v_j)$$
$$\vec{\mathrm{sup}}_{k+1} = \vec{\mathrm{sup}}_k - \vec{U}^{k+1} \text{ avec } U_j^{k+1} = \max_{(\vec{v} \in R^{k+1})}(v_j)$$

□

3.2 Fusion

Nous avons rappelé dans l'introduction de ce chapitre les conditions de validité de l'application de la fusion sur deux nids de boucles. Ces conditions sont données dans la *définition 10.*

Définition 10 *La fusion de deux nids de boucles est valide si et seulement si :*

1. *dans le nid fusionné, nous ne créons pas de l'instruction du deuxième nid vers l'instruction du premier nid une* dépendance de flot, *une* anti-dépendance *ou une* dépendance de sortie*;*
2. *ces deux nids doivent avoir le même domaine d'itération.*

Cette deuxième condition, nous pouvons l'ignorer de la manière suivante :

1. le domaine d'itération du nid fusionné sera l'union des domaines des deux nids ;
2. l'instruction de chaque *nid k* sera gardée par une condition pour qu'elle ne s'exécute que pour les itérations de son domaine D_k.

Exemple :

Soit la séquence de deux nids de boucles donnée en *figure 3.4*. Les deux nids de cette séquence n'ont pas le même domaine d'itération. Mais nous pouvons les fusionner en un seul nid qui est donné en *figure 3.5*.

```
do i = 1, 20
   a(i) = i        (* s1 *)
enddo
do i = 3, 10
   b(i) = a(i)     (* s2 *)
enddo
```

Figure 3.4: Exemple de code constitué de deux nids qui n'ont pas le même domaine d'itération

– Le domaine d'itération du nid fusionné sera donné par :

$$D = \{i:\ 1 \le i \le 20\} \cup \{i:\ 3 \le i \le 10\} = \{i:\ 1 \le i \le 20\}$$

– La garde sur l'instruction S_2 est donnée par if $(3 \le i \le 10)$.

```
do i = 1, 20
   a(i) = i                      (* s1 *)
   if (3 ≤ i ≤ 10) b(i) = a(i)   (* s2 *)
enddo
```

Figure 3.5: Fusion des deux nids du code en *figure 3.4*.

La garde sur l'instruction S_2 peut être éliminée en décomposant le domaine $D = \{i : 1 \le i \le 20\}$ en 3 sous domaines : $D_1 = \{i :\ 1 \le i \le 2\}$, $D_2 = \{i :\ 3 \le i \le 10\}$ et $D_3 = \{i :\ 11 \le i \le 20\}$. Cette décomposition peut se faire avec l'algorithme de Quilleré [63] ou par notre algorithme présenté dans le *chapitre 6* [6]. Après cette décomposition, nous obtenons le code en *figure 3.6*.

Comme nous avons contourné la deuxième condition de légalité de la fusion, alors nous pouvons appliquer la fusion sur le code en *figure 3.2*, en garantissant uniquement la première condition. Cette première condition est équivalente à assurer que tous les éléments du tableau A_k nécessaires pour le calcul d'un élément du tableau A_{k+1} à une itération $\vec{i}$ dans le nid fusionné ont déjà été calculés par les itérations précédentes. Pour satisfaire cette condition, nous translatons le domaine d'itération D_k de chaque nid avec un délai $\vec{h}_k$.

3.2.1 Translation des différents domaines d'itération

La translation est une technique de transformation de programmes qui est appelée dans les autres travaux *décalage de boucle* [12, 24] (*loop shifting*). Cette technique consiste à décaler

```
do i = 1, 2
   a(i) = i         (* s1 *)
enddo
do i = 3, 10
   a(i) = i         (* s1 *)
   b(i) = a(i)      (* s2 *)
enddo
do i = 11, 20
   a(i) = i         (* s1 *)
enddo
```

Figure 3.6: Élimination de la garde de l'instruction S_2 dans le code en *figure 3.5*

toutes les itérations d'un nid avec le même vecteur. L'effet de cette technique est d'avancer ou de retarder le calcul d'un élément d'un tableau. Soit Time_k la translation associée au *nid* k et définie de la manière suivante :

$$\begin{array}{rcccl} & & D_k & \longrightarrow & \mathbb{Z}^n \\ \mathrm{Time}_k & : & \vec{i}_k & \longmapsto & \vec{i} = \vec{i}_k + \vec{h}_k \end{array}$$

Toutes les itérations du domaine D_k seront décalées dans le domaine D'_k, c'est-à-dire que :

$$\begin{array}{rcl} D'_k & = & \{\vec{i} = \vec{i}_k + \vec{h}_k \ : \vec{\inf}_k \leq \vec{i}_k \leq \vec{\sup}_k\} \\ & = & \{\vec{i} \ : \vec{\inf}_k \leq \vec{i} - \vec{h}_k \leq \vec{\sup}_k\} \\ & = & \{\vec{i} : \vec{\inf}_k + \vec{h}_k \leq \vec{i} \leq \vec{\sup}_k + \vec{h}_k\} \end{array}$$

Cela signifie que les bornes inférieure et supérieure du domaine D'_k sont respectivement $\vec{\inf}_k + \vec{h}_k$ et $\vec{\sup}_k + \vec{h}_k$. Après la translation des différents nids du code en *figure 3.2*, nous obtenons le code en *figure 3.7*.

$$\begin{array}{ll}
\mathsf{do}\ \vec{i} \in D'_1 & \\
\quad A_1(\vec{i} - \vec{h}_1) = A_0(\vec{i}_1 - \vec{h}_1 + \vec{v}^1_1) \otimes .... \otimes A_0(\vec{i} - \vec{h}_1 + \vec{v}^1_{m_1}) & (*\ s_1\ *) \\
\mathsf{enddo} & \\
. & \\
. & \\
\mathsf{do}\ \vec{i} \in D'_k & \\
\quad A_k(\vec{i} - \vec{h}_k) = A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_1) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_{m_k}) & (*\ s_k\ *) \\
\mathsf{enddo} & \\
. & \\
. & \\
\mathsf{do}\ \vec{i} \in D'_p & \\
\quad A_p(\vec{i} - \vec{h}_p) = A_{p-1}(\vec{i} - \vec{h}_p + \vec{v}^p_1) \otimes .... \otimes A_{p-1}(\vec{i} - \vec{h}_p + \vec{v}^p_{m_p}) & (*\ s_p\ *) \\
\mathsf{enddo} &
\end{array}$$

Figure 3.7: Le code obtenu après la translation des domaines d'itération du code de la *figure 3.2*

Dans le reste de ce chapitre, nous notons par $\vec{O}_k$ l'origine du domaine D'_k et $(\vec{N}_k+\vec{O}_k-\vec{1})$ sa borne supérieure, c'est à dire que $D'_k = \{\vec{i}\ |\vec{O}_k \leq \vec{i} \leq \vec{N}_k + \vec{O}_k - \vec{1}\}$.

3.2.2 Condition de légalité de la fusion

Les nids du code en *figure 3.7* n'ont pas le même domaine d'itération. Pour contourner la deuxième condition de validité de l'application de la fusion, le domaine d'itération de leur fusion (*figure 3.8*) sera donné par l'union des translations des domaines $D_{k=1..p}$ par les vecteurs $\vec{h}_{k=1..p}$:

$$D_{iter} = \cup_{k=1}^{p}(D'_k) \text{ avec } D'_k = \{\vec{i}_k + \vec{h}_k \mid \vec{i}_k \in D_k\}$$

Ce domaine n'étant pas nécessairement convexe, nous utiliserons son enveloppe convexe pour générer le code. Comme l'instruction S_k ne sera pas exécutée pour toute itération du domaine D_{iter} dans le nid fusionné alors elle sera gardée par la condition :

$$C_k(\vec{i}) = (\vec{i} \in D'_k)$$

Comme nous avons contourné la deuxième condition de légalité de la fusion et que les codes considérés dans cette thèse sont particuliers, la condition de validité de la fusion des nids du code en *figure 3.7* en un seul nid du code en *figure 3.8* sera donnée par le *lemme 1.*

Lemme 1 *La fusion des nids du code en* figure 3.2 *en un seul nid du code en* figure 3.8 *est valide si et seulement si dans le code en* figure 3.8, *nous ne créons pas de l'instruction S_{k+1} vers l'instruction S_k une* anti-dépendance.

Preuve : Découle du fait que nous avons contourné la deuxième condition de la *définition 10* et que les *dépendances de flot* et les *dépendances de sortie* ne peuvent pas exister dans le code *figure 3.8* de l'instruction S_{k+1} vers l'instruction S_k.

□

do $\vec{i} \in D_{iter}$
  $C_1(\vec{i})\ A_1(\vec{i}-\vec{h}_1) = A_0(\vec{i}-\vec{h}_1+\vec{v}_1^1) \otimes .... \otimes A_0(\vec{i}-\vec{h}_1+\vec{v}_{m_1}^1)$ (* s_1 *)
  .
  .
  $C_k(\vec{i})\ A_k(\vec{i}-\vec{h}_k) = A_{k-1}(\vec{i}-\vec{h}_k+\vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i}-\vec{h}_k+\vec{v}_{m_k}^k)$ (* s_k *)

  $C_p(\vec{i})\ A_p(\vec{i}-\vec{h}_p) = A_{p-1}(\vec{i}-\vec{h}_p+\vec{v}_1^p) \otimes .... \otimes A_{p-1}(\vec{i}-\vec{h}_p+\vec{v}_{m_p}^p)$ (* s_p *)
enddo

Figure 3.8: Fusion des nids du code de la *figure 3.2* après la translation des différents domaines et introduction des gardes

Lemme 2 *La fusion de tous les nids du code de la* figure 3.2 *en un seul nid qui est donné en* figure 3.8 *est valide si et seulement si :*

$$\forall \vec{i}_k \in D_k,\ \forall \vec{i}_{k+1} \in D_{k+1} :\ (\exists \vec{v} \in R^{k+1}\ : \vec{i}_k = \vec{i}_{k+1} + \vec{v}) \Rightarrow Time_k(\vec{i}_k) \preceq Time_{k+1}(\vec{i}_{k+1})$$

Preuve :

$\Rightarrow$? Nous supposons que :

- (a) notre fusion est valide et
- (b) $\exists \vec{i_k} \in D_k$, $\exists \vec{i}_{k+1} \in D_{k+1}$: $(\exists \vec{v} \in R^{k+1} : \vec{i_k} = \vec{i}_{k+1} + \vec{v}) \wedge (\mathrm{Time}_k(\vec{i_k}) \succ \mathrm{Time}_{k+1}(\vec{i}_{k+1}))$.

Dans le code en *figure 3.2* :

1. à l'itération $\vec{i_k} \in D_k$ nous calculons $A_k(\vec{i_k})$;
2. à l'itération $\vec{i}_{k+1} \in D_{k+1}$ nous utilisons $A_k(\vec{i}_{k+1} + \vec{v})$.

$$\begin{array}{rcl} \vec{i_k} = \vec{i}_{k+1} + \vec{v} & \Rightarrow & \text{l'élément du tableau } A_k \text{ produit par } \vec{i_k} \text{ sera consommé par } \vec{i}_{k+1}. \\ \text{Soient } \mathrm{Time}_k(\vec{i_k}) & = & \vec{i} \in D'_k \text{ et } \mathrm{Time}_{k+1}(\vec{i}_{k+1}) = \vec{i'} \in D'_{k+1} \\ \mathrm{Time}_k(\vec{i_k}) & \succ & \mathrm{Time}_{k+1}(\vec{i}_{k+1}) \Rightarrow \vec{i} \succ \vec{i'} \\ & \Rightarrow & \text{dans le code en } \textit{figure 3.8}, \ \vec{i} \text{ sera exécutée après l'itération } \vec{i'}. \\ & & \text{Comme } \vec{i} \text{ produit un élément qui sera consommé par } \vec{i'}, \\ & & \text{alors nous créons une } \textit{anti-dépendance} \text{ de l'instruction } S_{k+1} \\ & & \text{vers l'instruction } S_k. \text{ Contradiction avec l'hypothèse (a) } (\textit{lemme 1}). \end{array}$$

$\Leftarrow$? Nous supposons que :

- (a) $\forall \vec{i_k} \in D_k$, $\forall \vec{i}_{k+1} \in D_{k+1}$: $(\exists \vec{v} \in R^{k+1} : \vec{i_k} = \vec{i}_{k+1} + \vec{v}) \Rightarrow \mathrm{Time}_k(\vec{i_k}) \preceq \mathrm{Time}_{k+1}(\vec{i}_{k+1})$ et
- (b) notre fusion n'est pas valide.

D'après le *lemme 1*, la fusion peut être non valide dans un seul cas : il existe une *anti-dépendance* d'une instruction S_{k+1} vers l'instruction S_k dans le code en *figure 3.8*. C'est-à-dire que :

$$\exists \vec{i} \in D'_k, \exists \vec{i'} \in D'_{k+1} : (\exists \vec{v} \in R^{k+1} : \vec{i} - \vec{h}_k = \vec{i'} - \vec{h}_{k+1} + \vec{v}) \text{ et } (\vec{i} \succ \vec{i'})$$

Cela signifie que :

$$\exists \vec{i_k} \in D_k, \exists \vec{i}_{k+1} \in D_{k+1} : (\exists \vec{v} \in R^{k+1} : \vec{i_k} = \vec{i}_{k+1} + \vec{v}) \text{ et } \mathrm{Time}_k(\vec{i_k}) \succ \mathrm{Time}_{k+1}(\vec{i}_{k+1})$$

Ceci est contradictoire avec *l'hypothèse (a).*

□

Théorème 1 *La condition de légalité de la fusion donnée dans le* lemme 2 *est équivalente à :*

$$\forall k \in [1 \cdots p-1] \ : \ -\vec{h}_k \succeq -\vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}}$$

Preuve :

$\Rightarrow$? Nous supposons que :

- (a) notre fusion est valide et
- (b) $\exists k :\ -\vec{h}_k \prec -\vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}}$.

À une itération $\vec{i} \in D'_{k+1}$ dans le code fusionné de la *figure 3.8*, pour calculer $A_{k+1}(\vec{i} - \vec{h}_{k+1})$ nous avons besoin de $A_k(\vec{i} - \vec{h}_{k+1} + \vec{v}^{k}_{m_{k+1}})$ qui est calculé à son tour dans le domaine D'_k à l'itération $\vec{i}_{m_{k+1}} = \vec{i} - \vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}} + \vec{h}_k$.

$$\begin{aligned} (b) \quad &\Rightarrow \quad \vec{i} - \vec{h}_k \prec \vec{i} - \vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}} \\ &\Rightarrow \quad \vec{i} \prec \vec{i} - \vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}} + \vec{h}_k \\ &\Rightarrow \quad \vec{i} \prec \vec{i}_{m_{k+1}} \end{aligned}$$

Cela signifie qu'il existe une *anti-dépendance* de l'instruction S_{k+1} vers l'instruction S_k dans le code en *figure 3.8.* Contradiction avec l'hypothèse (a).

$\Leftarrow$? Nous supposons que :

- (a) $\forall k : -\vec{h}_k \succeq -\vec{h}_{k+1} + \vec{v}^{k+1}_{m_{k+1}}$ et
- (b) notre fusion n'est pas valide.

D'après le *lemme 2*, la fusion n'est pas valide, alors :

$$\exists \vec{i}_k \in D_k,\ \exists \vec{i}_{k+1} \in D_{k+1} :\ (\exists \vec{v} \in R^{k+1}\ : \vec{i}_k = \vec{i}_{k+1} + \vec{v}) \wedge \mathrm{Time}_k(\vec{i}_k) \succ \mathrm{Time}_{k+1}(\vec{i}_{k+1})$$

Cela signifie que :

$$\exists \vec{i}_k \in D_k,\ \exists \vec{i}_{k+1} \in D_{k+1} :\ (\exists \vec{v} \in R^{k+1}\ : \vec{i}_k = \vec{i}_{k+1} + \vec{v}) \wedge (\vec{i}_k + \vec{h}_k \succ \vec{i}_{k+1} + \vec{h}_{k+1})$$

Donc :

$$\exists \vec{v} \in R^{k+1} :\ \vec{v} \succ \vec{h}_{k+1} - \vec{h}_k.$$

$$(a) \Rightarrow \quad \vec{v}^{k+1}_{m_{k+1}} \preceq \vec{h}_{k+1} - \vec{h}_k.$$

Cela signifie que $\vec{v} \succ \vec{v}^{k+1}_{m_{k+1}}$. Contradiction car $\forall \vec{v} \in R^{k+1} :\ \vec{v} \preceq \vec{v}^{k+1}_{m_{k+1}}$.

□

3.3 Fusion avec allocation des tampons

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les mêmes données, nous remplaçons les tableaux A_1, A_2,...et A_{p-1} par des tampons B_1, B_2,...et B_{p-1}. Un tampon B_k est un tableau à une ou plusieurs dimensions qui contiendra les données vivantes du tableau A_k.

3.3.1 Données vivantes

Définition 11 (Donnée vivante) *Une donnée du tableau A_{k-1} est dite vivante à une itération $\vec{i} \in D'_k$ si et seulement si :*

1. *elle est produite dans le domaine D'_{k-1} à une itération $\vec{i}_1 \preceq \vec{i}$.*
2. *il existe dans le domaine D'_k une itération $\vec{i}_2 \succeq \vec{i}$ qui la consommera.*

Définition 12 (Volume mémoire) *Le volume mémoire $M_k(\vec{i})$ correspondant à une itération $\vec{i} \in D'_{k=2,n}$ est le nombre d'éléments du tableau A_{k-1} qui sont vivants :*

$$\begin{array}{rcl} M_k(\vec{i}) & = & |E_k(\vec{i})| \text{ avec} \\ E_k(\vec{i}) & = & \{\vec{i}_1 \in D'_{k-1} | \exists \vec{v} \in R^k \text{ et } \exists \ \vec{i}_2 \in D'_k \ : \vec{i}_1 - \vec{h}_{k-1} = \vec{i}_2 - \vec{h}_k + \vec{v} \text{ et } \vec{i}_1 \preceq \vec{i} \preceq \vec{i}_2\} \end{array}$$

Définition 13 (Successeur) *Le successeur de l'itération $\vec{i} = (i_1, \cdots, i_n)^t$ dans le domaine D'_k sera noté par $succ(\vec{i})$ et il est défini comme suit :*

$$succ(\vec{i}) = \begin{cases} \vec{i} + (0, \cdots, 0, 1)^t & si \ \ i_n \neq N_{k,n} + O_{k,n} - 1 \\ (succ((i_1, \cdots, i_{n-1})^t), \ O_{k,n})^t & sinon \end{cases}$$

La *définition 13* signifie que si $i_n = N_{k,n} + O_{k,n} - 1$, alors nous devons chercher le plus petit des m tel que $\forall j \in [m \cdots n] \ : i_j = N_{k,j} + O_{k,j} - 1$. Dans ce cas le successeur de $\vec{i}$ sera donné par :

$$\begin{array}{rcl} \text{succ}(\vec{i}) & = & \vec{i} + \vec{\text{Inc}}_{k,m} \\ \vec{\text{Inc}}_{k,m} & = & (0, \ 0, \cdots, \ 0, \ 1, \ -N_{k,m} + 1, \cdots, \ -N_{k,n} + 1)^t \end{array}$$

Proposition 2 *Soit $\vec{C}_k = (\prod_{i=2}^{n}(N_{k,i}), \cdots, \prod_{i=n}^{n}(N_{k,i}), \ 1)^t$. Le produit scalaire entre $\vec{Inc}_{k,m}$ et $\vec{C}_k$ est égal à 1 :*

$$\forall m \in [1 \cdots n] : \ \vec{Inc}_{k,m} \ . \ \vec{C}_k = 1.$$

Preuve :

$$\begin{array}{rcl} \vec{\text{Inc}}_{k,m} \ . \ \vec{C}_k & = & (0, \ 0, \cdots, \ 0, \ 1, \ -N_{k,m} + 1, \cdots, \ -N_{k,n} + 1)^t . \ (\prod_{i=2}^{n}(N_{k,i}), \cdots, \ \prod_{i=n}^{n}(N_{k,i}), \ 1)^t . \\ & = & \prod_{i=m}^{n}(N_{k,i}) + (-N_{k,m} + 1) \prod_{i=m+1}^{n}(N_{k,i}) + (-N_{k,m+1} + 1) \prod_{i=m+2}^{n}(N_{k,i}) + \\ & & \cdots + (-N_{k,n-1} + 1) \prod_{i=n}^{n}(N_{k,i}) - N_{k,n} + 1 \\ & = & \prod_{i=m}^{n}(N_{k,i}) - \prod_{i=m}^{n}(N_{k,i}) + \prod_{i=m+1}^{n}(N_{k,i}) - \prod_{i=m+1}^{n}(N_{k,i}) + \cdots + N_{k,n} - N_{k,n} + 1 \\ & = & 1 \end{array}$$

□

Lemme 3 *Soient $\vec{i}$ une itération dans le domaine D'_k et $nbr_k(\vec{O}_k, \ \vec{i})$ le nombre d'itérations dans D'_k qui sont comprises entre $\vec{O}_k$ et $\vec{i}$; $nbr_k(\vec{O}_k, \ \vec{i}) = |\{\vec{i'} \in D'_k : \ \vec{O}_k \preceq \vec{i'} \preceq \vec{i}\}|$. Ce nombre est donné par :*

$$\begin{array}{rcl} & & \vec{C}_k . \ (\vec{i} - \vec{O}_k) + 1 \text{ avec} \\ \vec{C}_k & = & (\prod_{i=2}^{n} N_{k,i}, \cdots, \prod_{i=n}^{n} N_{k,i}, \ 1)^t \end{array}$$

Preuve : Nous notons par $P(\vec{i})$ cette propriété et nous montrons par récurrence que $\forall \vec{i} \in D'_k$: $P(\vec{i})$ est vraie.

1. $P(\vec{O}_k)$ est vraie ?
   $\text{nbr}_k(\vec{O}_k,\ \vec{O}_k) = (\vec{O}_k - \vec{O}_k)\ .\vec{C}_k + 1 = 1$. Par conséquent $P(\vec{O}_k)$ est varie.
2. Nous supposons que $P(\vec{i})$ est vraie et nous montrons que $P(\text{succ}(\vec{i}))$ l'est aussi ?

   Nous savons que $\text{succ}(\vec{i}) = \vec{i} + \vec{\text{Inc}}_k$ avec $\vec{\text{Inc}}_k = \begin{cases} (0, \cdots, 0, 1)^t & \text{si } i_n \neq N_{k,n} + O_{k,n} - 1 \\ \vec{\text{Inc}}_{k,m} \text{ sinon} \end{cases}$

   (a) $\vec{\text{Inc}}_{k,m}\ .\ \vec{C}_k = 1$ (*proposition 2*).

   (b) nous pouvons vérifier facilement que $(0,\ \cdots,\ 0,\ 1)^t\ .\ \vec{C}_k = 1$.

   (a) et (b) $\Rightarrow \vec{\text{Inc}}_k\ .\vec{C}_k = 1$.
   Comme il n'y a pas d'itérations entre $\vec{i}$ et $\text{succ}(\vec{i})$ alors

   $$\text{nbr}_k(\vec{O}_k,\ \text{succ}(\vec{i})) = \text{nbr}_k(\vec{O}_k,\ \vec{i}) + 1$$

   Comme $\vec{\text{Inc}}_k\ .\ \vec{C}_k = 1$ alors :

   $$\text{nbr}_k(\vec{O}_k,\ \text{succ}(\vec{i})) = \text{nbr}_k(\vec{O}_k,\ \vec{i}) + \vec{\text{Inc}}_k\ .\vec{C}_k$$

   Nous avons par hypothèse $\text{nbr}_k(\vec{O}_k,\ \vec{i}) = (\vec{i} - \vec{O}_k)\ .\ \vec{C}_k + 1$. Par conséquent :

   $$\begin{aligned} \text{nbr}_k(\vec{O}_k,\ \text{succ}(\vec{i})) &= (\vec{i} - \vec{O}_k)\ .\ \vec{C}_k + 1 + \vec{\text{Inc}}_k\ .\vec{C}_k \\ &= (\vec{i} + \vec{\text{Inc}}_k - \vec{O}_k)\ .\ \vec{C}_k + 1 \\ &= (\text{succ}(\vec{i}) - \vec{O}_k)\ .\ \vec{C}_k + 1 \end{aligned}$$

   Par conséquent $P(\text{succ}(\vec{i}))$ est vraie.

□

Lemme 4 *Soient $\vec{i}_1$ et $\vec{i}_2$ ($\vec{i}_1 \preceq \vec{i}_2$) deux itérations dans le domaine D'_k et $\text{nbr}_k(\vec{i}_1,\ \vec{i}_2)$ le nombre d'itérations dans D'_k qui sont comprises entre $\vec{i}_1$ et $\vec{i}_2$: $\text{nbr}_k(\vec{i}_1,\ \vec{i}_2) = |\{\vec{i}' \in D'_k : \vec{i}_1 \preceq \vec{i}' \preceq \vec{i}_2\}$. Ce nombre est donné par :*

$$\vec{C}_k.\ (\vec{i}_2 - \vec{i}_1) + 1$$

Preuve : Nous savons que :

$$\begin{aligned} \text{nbr}_k(\vec{i}_1,\ \vec{i}_2) &= \text{nbr}_k(\vec{O}_k,\ \vec{i}_2) - \text{nbr}_k(\vec{O}_k,\ \vec{i}_1) + 1 \\ &= ((\vec{i}_2 - \vec{O}_k)\ .\vec{C}_k + 1) - (\vec{i}_1 - \vec{O}_k)\ .\vec{C}_k + 1) + 1 \\ &= (\vec{i}_2 - \vec{i}_1)\ .\ \vec{C}_k + 1 \end{aligned}$$

□

Lemme 5 *Soient* $\vec{i} \in D'_k$ *et* $\vec{i}_1 = \vec{i} - (\vec{h}_k - \vec{h}_{k-1} - \vec{v}_1^k) \in D'_{k-1}$. *Aucune donnée du tableau* A_{k-1} *produite avant* $\vec{i}_1$ *n'est vivante. C'est à dire que :*

$$\forall \vec{j}_1 \prec \vec{i}_1, \forall \vec{v}_q^k \in R^k,\ \forall \vec{j}_2 \succeq \vec{i} : \vec{j}_1 - \vec{h}_{k-1} \neq \vec{j}_2 - \vec{h}_k + \vec{v}_q^k.$$

Preuve : Nous supposons qu'il existe une donnée produite avant $\vec{i}_1$ et qui est vivante à l'itération $\vec{i}$, c'est-à-dire que :

$$\begin{aligned} \exists \vec{j}_1 \prec \vec{i}_1, \exists \vec{v}_q^k \in R^k,\ \exists \vec{j}_2 \succeq \vec{i} : \vec{j}_1 - \vec{h}_{k-1} = \vec{j}_2 - \vec{h}_k + \vec{v}_q^k \ &\Rightarrow\ \vec{j}_1 = \vec{j}_2 - \vec{h}_k + \vec{v}_q^k + \vec{h}_{k-1} \\ &\Rightarrow\ \vec{i}_1 - \vec{j}_1 = \vec{i} - \vec{j}_2 + (\vec{v}_1^k - \vec{v}_q^k) \end{aligned}$$

$$\begin{aligned} \vec{j}_1 \prec \vec{i}_1 \ &\Rightarrow\ \vec{i} - \vec{j}_2 + (\vec{v}_1^k - \vec{v}_q^k) \succ \vec{0} \\ \vec{j}_2 \succeq \vec{i} \ &\Rightarrow\ (\vec{v}_1^k - \vec{v}_q^k) \succ \vec{0} \\ &\Rightarrow\ (\vec{v}_1^k \succ \vec{v}_q^k) \end{aligned}$$

Contradiction, car nous avons par hypothèse l'ordre suivant $\vec{v}_1^k \preceq \vec{v}_2^k \preceq \ldots.. \preceq \vec{v}_{m_k}^k$.

□

Théorème 2 *Le volume mémoire* $M_k(\vec{i} \in D'_k)$ *est borné par* $Sup_k = \vec{C}_{k-1}.\ (\vec{h}_k - \vec{h}_{k-1} - \vec{v}_1^k) + 1$.

Preuve : À l'itération $\vec{i} \in D'_k$ pour calculer $A_k(\vec{i} - \vec{h}_k)$ nous utilisons m_k données du tableau A_{k-1} qui sont produites respectivement par $\vec{i}_1$, $\vec{i}_2$,... et $\vec{i}_{m_k}$ telles que $\vec{i}_q = \vec{i} - (\vec{h}_k - \vec{h}_{k-1} - \vec{v}_q^k)$ $(q = 1 \cdots m_k)$. Nous savons que :
- (a) $\vec{i}_1 \preceq \vec{i}_2 \preceq \cdots \preceq \vec{i}_{m_k}$;
- (b) la donnée produite par $\vec{i}_1$ est vivante car $\vec{i}_1 \preceq \vec{i}$ et elle est consommée par $\vec{i}$;
- (c) aucune donnée produite avant $\vec{i}_1$ n'est vivante (*lemme 5*).

(a), (b) et (c) signifient que le volume mémoire $M_k(\vec{i})$ est borné par le nombre d'itérations dans D'_{k-1} qui sont comprises entre $\vec{i}_1$ et $\vec{i}$. D'après le *lemme 4* cette borne sera donnée par :

$$\begin{aligned} \mathrm{Sup}_k &= \mathrm{nbr}_k(\vec{i}_1,\ \vec{i}) \\ &= \vec{C}_{k-1}\ .\ (\vec{i} - \vec{i}_1) + 1 \\ &= \vec{C}_{k-1}.\ (\vec{h}_k - \vec{h}_{k-1} - \vec{v}_1^k) + 1 \end{aligned}$$

□

Nous pouvons vérifier facilement que ce volume est toujours positif.

3.3.2 Allocation des tampons

Comme nous l'avons mentionné précédemment, chaque tableau A_k sera remplacé par un tampon B_k qui contiendra les données vivantes. Le tampon B_k peut avoir une ou plusieurs dimensions et il peut être géré de plusieurs façons.

Tampons à une seule dimension :

Dans ce cas, le tampon B_k va avoir une seule dimension. Il y a plusieurs techniques pour gérer ce tampon :

1. À chaque nouvelle itération $\vec{i} \in D_k$, tous les éléments du tampon B_k seront décalés vers la gauche et la nouvelle donnée du tableau A_k produite par $\vec{i}$ sera stockée dans la dernière case du tampon B_k.
2. La nouvelle donnée du tableau A_k produite à l'itération $\vec{i}$ va écraser la donnée morte. C'est-à-dire que le tampon B_k sera géré d'une manière séquentielle et circulaire.

Les décalages successifs des éléments du tampon B_k sont très coûteux, ce qui rend la première technique inefficace. Par conséquent, dans ce chapitre nous utiliserons la deuxième technique pour gérer les différents tampons B_k, c'est-à-dire que chaque tampon B_k sera géré d'une manière circulaire et séquentielle.

Tampons à plusieurs dimensions :

Dans ce cas le tampon B_k va avoir plusieurs dimensions. De la même manière que le cas à une dimension, nous pouvons gérer de plusieurs façons le tampon B_k. Du point de vue localité, cette solution n'apporte pas un plus par rapport aux tampons à une dimension.

3.3.3 Tampon séquentiel et circulaire à une dimension

Dans ce cas chaque tampon B_k sera à une seule dimension et il sera géré d'une manière séquentielle et circulaire :

1. le fait que le tampon B_k est séquentiel signifie que l'élément produit par $\mathrm{succ}(\vec{i}) \in D'_k$ sera rangé juste à coté de l'élément qui est produit par $\vec{i} \in D'_k$.
2. le fait que le tampon B_k est circulaire signifie que si $\vec{i} \in D'_k$ range un élément dans la dernière case alors l'élément produit par $\mathrm{succ}(\vec{i}) \in D'_k$ sera rangé dans la première case du tampon B_k.

Pour ranger et récupérer les éléments du tableau A_k dans le tampon B_k, nous lui associons une fonction d'accès F_k :

$$\begin{array}{rccc} & D'_k & \longrightarrow & N \\ F_k \ : & \vec{i} & \longmapsto & F_k(\vec{i}) \end{array}$$

Nous notons T_k la taille du tampon B_k et nous supposons que les adresses des cases de ce tampon varient de 0 à $T_k - 1$. Pour commencer le rangement à la première case et avoir un stockage séquentiel et circulaire, la fonction F_k doit satisfaire aux deux conditions suivantes :

$$\begin{array}{rcl} F_k(\vec{O}_k) & = & 0 \\ F_k(\mathrm{succ}(\vec{i})) & = & \begin{cases} F_k(\vec{i}) + 1 \text{ si } (F_k(\vec{i}) \neq \mathrm{T}_k - 1) \\ 0 \qquad\qquad \text{sinon} \end{cases} \end{array}$$

Proposition 3 *Si deux itérations distinctes $\vec{i}$ et $\vec{i'}$ du domaine D'_k écrivent dans la même case du tampon B_k, alors le nombre d'itérations qui sont comprises entre $\vec{i}$ et $\vec{i'}$ (ces deux itérations incluses) est supérieur à la taille du tampon B_k.*

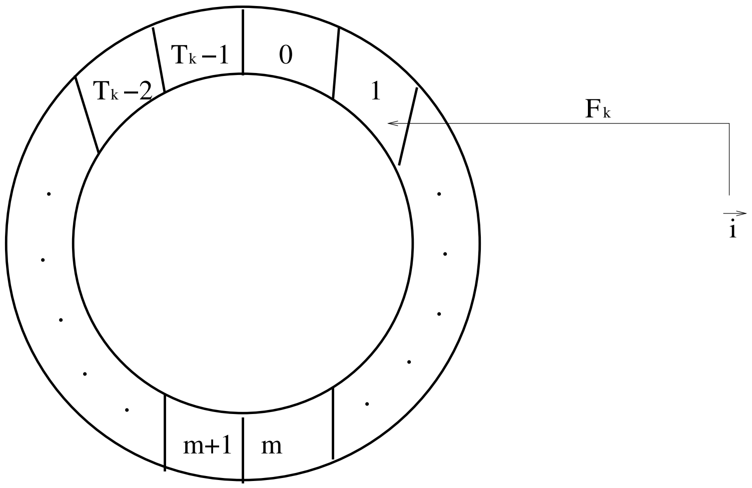


Figure 3.9: Tampon séquentiel et circulaire

Preuve : La preuve est évidente ; elle découle du fait que les écritures dans ce tampon sont séquentielles et que ce tampon est circulaire.

□

Soit B_k le tampon circulaire et séquentiel qui est associé au tableau A_k. Après le remplacement du tableau A_k par le tampon B_k dans le code généré, nous devons nous assurer que nous n'écrasons pas de données vivantes du tableau A_k dans le tampon B_k.

Proposition 4 *Soit $Sup_{k+1} = \vec{C}_k \ . \ (\vec{h}_{k+1} - \vec{h}_k - \vec{v}_1^{k+1}) + 1$ la taille du tampon B_k. Quelle que soit l'itération $\vec{i} \in D'_{k+1}$, toutes ses données vivantes sont dans le tampon B_k.*

Preuve : Nous supposons qu'il existe une itération $\vec{i} \in D_{k+1}$ telle que l'une de ses données vivantes ne se trouve pas dans le tampon B_k. Par définition, cette donnée vivante est produite par $\vec{i}_1 \preceq \vec{i}$ et elle sera consommée par $\vec{i}_2 \succeq \vec{i}$. Comme $\vec{i}_1 \preceq \vec{i}$ alors cette donnée vivante a été déjà rangée dans le tampon B_k et il existe nécessairement une itération $\vec{i}_3$ ($\vec{i}_1 \prec \vec{i}_3 \prec \vec{i}$) qui l'a écrasée. Dans un tampon séquentiel et circulaire, si deux itérations écrivent dans la même case, alors le nombre d'itérations qui séparent ces deux itérations est supérieur à la taille du tampon, c'est-à-dire que le nombre d'itérations entre $\vec{i}_1$ et $\vec{i}_3$ est supérieur à Sup_{k+1}. Par conséquent, le nombre d'itérations entre $\vec{i}_2$ et l'une de ses productions est aussi supérieur à Sup_{k+1}. Contradiction, car le nombre d'itérations entre une itération $\vec{i} \in D'_{k+1}$ et la plus ancienne de ses productions est borné par Sup_{k+1}.

□

Comme nous l'avons mentionné précédemment, pour stocker et récupérer les éléments du tableau A_k dans le tampon B_k, nous lui associons une fonction d'accès F_k :

$$F_k : \begin{array}{ccc} D'_k & \longrightarrow & N \\ \vec{i} & \longmapsto & F_k(\vec{i}) \end{array}$$

3.3.4 Fonction d'accès

Comme la taille du tampon B_k est Sup_{k+1}, pour commencer le rangement à la première case et avoir un stockage séquentiel et circulaire, la fonction F_k doit satisfaire :

$$F_k(\vec{O}_k) = 0 \tag{3.2}$$

$$F_k(\mathrm{succ}(\vec{i})) = \begin{cases} F_k(\vec{i}) + 1 \text{ si } (F_k(\vec{i}) \neq \mathrm{Sup}_{k+1} - 1) \\ 0 \qquad\qquad \text{sinon} \end{cases} \tag{3.3}$$

La proposition suivante donne l'un des choix possible de la fonction F_k pour que les conditions 3.2 et 3.3 soient satisfaites.

Proposition 5 *Pour satisfaire les conditions 3.2 et 3.3 on peut choisir :*
$F_k(\vec{i}) = [(\vec{i} - \vec{O}_k)\ .\ \vec{C}_k] \mod Sup_{k+1}$.

Preuve :

1. Condition 3.2 :
   $F_k(\vec{O}_k) = ((\vec{O}_k - \vec{O}_k).\vec{C}_k) \mod \mathrm{Sup}_{k+1} = (\vec{O}\ .\ \vec{C}_k) \mod \mathrm{Sup}_{k+1} = 0$
2. Condition 3.3 :
   Soit $F_k(\vec{i}) = H_k(\vec{i}) \mod \mathrm{Sup}_{k+1}$ avec $H_k(\vec{i}) = (\vec{i} - \vec{O}_k)\ .\ \vec{C}_k$. Pour satisfaire la condition 3.3, il suffit que :

$$H_k(\mathrm{succ}(\vec{i})) = H_k(\vec{i}) + 1$$

Nous savons que $\mathrm{succ}(\vec{i}) = \vec{i} + \vec{\mathrm{Inc}}_k$ avec $\vec{\mathrm{Inc}}_k = \begin{cases} (0, \cdots, 0, 1)^t & \text{si } i_n \neq N_{k,n} + O_{k,n} - 1 \\ \vec{\mathrm{Inc}}_{k,m} & \text{sinon} \end{cases}$

(a) $\vec{Inc}_k = (0, \cdots, 0, 1)^t$:

$$\begin{aligned} H_k(\mathrm{succ}(\vec{i})) &= (\vec{i} - \vec{O}_k + \vec{\mathrm{Inc}}_k)\ .\ \vec{C}_k \\ &= (\vec{i} - \vec{O}_k)\ .\ \vec{C}_k + \vec{\mathrm{Inc}}_k\ .\ \vec{C}_k \\ &= H_k(\vec{i}) + 1 \end{aligned}$$

(b) $\vec{\mathrm{Inc}}_k = \vec{\mathrm{Inc}}_{k,m}$:

$$\begin{aligned} H_k(\mathrm{succ}(\vec{i})) &= (\vec{i} - \vec{O}_k + \vec{\mathrm{Inc}}_{k,m})\ .\ \vec{C}_k \\ &= (\vec{i} - \vec{O}_k)\ .\ \vec{C}_k + \vec{\mathrm{Inc}}_{k,m}\ .\ \vec{C}_k \\ &= H_k(\vec{i}) + 1, \text{ car } \vec{\mathrm{Inc}}_{k,m}\ .\ \vec{C}_k = 1 \ (\textit{proposition 2}). \end{aligned}$$

□

3.3.5 Génération du code

Nous considérons l'instruction S_k du code fusionné de la *figure 3.8* qui est donnée par :

$$C_k(\vec{i})\ A_k(\vec{i} - \vec{h}_k) = A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}_{m_k}^k)$$

À chaque nouvelle itération $\vec{i} \in D'_k$, nous calculons l'élément $A_k(\vec{i} - \vec{h}_k)$ en fonction de m_k éléments du tableau A_{k-1} qui sont produits respectivement par :

$$\vec{i}_1, \vec{i}_2, \cdots \text{ et } \vec{i}_{m_k} \text{ avec } \vec{i}_q = \vec{i} - (\vec{h}_k - \vec{h}_{k-1} - \vec{v}_q^k)$$

1. L'élément $A_k(\vec{i}-\vec{h}_k)$ sera rangé dans le tampon B_k par l'itération $\vec{i}$ à la position $F_k(\vec{i})$.
2. Les m_k éléments du tableau A_{k-1} sont déjà rangés dans le tampon B_{k-1} par les itérations $\vec{i}_1$, $\vec{i}_2$, $\cdots$ et $\vec{i}_{m_k}$ respectivement aux positions $F_{k-1}(\vec{i}_1)$, $F_{k-1}(\vec{i}_2)$,$\cdots$ et $F_{k-1}(\vec{i}_{m_k})$.

Par conséquent l'instruction S_k sera remplacée par :

$$C_k(\vec{i})\ B_k(F_k(\vec{i})) = B_{k-1}(F_{k-1}(\vec{i}_1)) \otimes ... \otimes B_{k-1}(F_{k-1}(\vec{i}_{m_k}))$$

Le code fusionné après le remplacement des tableaux A_1,$\cdots$, A_{n-1} par des tampons B_1,$\cdots$, B_{n-1} est donné en *figure 3.10*.

do $\vec{i} \in D_{iter}$

$C_1(\vec{i})\ B_1(F_1(\vec{i})) = A_0(\vec{i}-\vec{h}_1+\vec{v}_1^1) \otimes .... \otimes A_0(\vec{i}-\vec{h}_1+\vec{v}_{m_1}^1)$ (* s_1 *)

.

.

$C_k(\vec{i})\ B_k(F_k(\vec{i})) = B_{k-1}(F_{k-1}(\vec{i}_1)) \otimes ... \otimes B_{k-1}(F_{k-1}(\vec{i}_{m_k}))$. (* s_k *)

$C_p(\vec{i})\ A_p(\vec{i}-\vec{h}_p) = B_{p-1}(F_{p-1}(\vec{i}_1)) \otimes ... \otimes B_{p-1}(F_{p-1}(\vec{i}_{m_p}))$. (* s_p *)

enddo

Figure 3.10: Fusion avec allocation des tampons à une dimension

3.4 Exemple complet

Ici, nous donnons un exemple de code constitué de deux nids de boucles. Avant de décrire l'application de la fusion et la fusion avec allocation des tampons sur ce code, nous montrons comment se fait la dérivation des domaines d'itérations des deux nids de ce code.

3.4.1 Calcul des domaines d'itérations

Nous considérons la séquence de nids de boucles donnée en *figure 3.11*. Les patrons de ces deux nids sont respectivement donnés par les ensembles R^1 et R^2 :

$$R^1 = R^2 = \{(-1,\ 0)^t,\ (0,\ -1)^t,\ (0,\ 0)^t,\ (0,\ 1)^t,\ (1,\ 0)^t\}$$

Le domaine de définition du tableau A_0 est imposé par l'utilisateur et il est défini par :

$$D_0 = \{\vec{i} = (i,\ j)^t :\ (1,\ 1)^t \le \vec{i} \le (N_{0,1}, N_{0,2})^t\}$$

Le domaine d'itération du premier nid est donné en fonction du domaine D_0 et de son patron. Il est dérivé de la manière suivante :

$$\begin{aligned} D_1 &= \{\vec{i} = (i,\ j)^t :\ (1,\ 1)^t - (-1,\ -1)^t \le \vec{i} \le (N_{0,1}, N_{0,2})^t - (1,\ 1)^t\} \\ &= \{\vec{i} = (i,\ j)^t :\ (2,\ 2)^t \le \vec{i} \le (N_{0,1}-1,\ N_{0,2}-1)^t\} \end{aligned}$$

Le domaine d'itération du deuxième nid est donné en fonction du domaine D_1 et de son patron. Il est dérivé de la manière suivante :

$$\begin{aligned} D_2 &= \{\vec{i} = (i,\ j)^t :\ (2,\ 2)^t - (-1,\ -1)^t \le \vec{i} \le (N_{0,1}-1, N_{0,2}-1)^t - (1,\ 1)^t\} \\ &= \{\vec{i} = (i,\ j)^t :\ (3,\ 3)^t \le \vec{i} \le (N_{0,1}-2,\ N_{0,2}-2)^t\} \end{aligned}$$

```
do i = 2, N_{0,1} - 1
   do j = 2, N_{0,2} - 1
      A_1(i, j) = A_0(i - 1, j) + A_0(i, j - 1)+
                  A_0(i, j) + A_0(i, j + 1) + A_0(i + 1, j)
   enddo
enddo
do i = 3, N_{0,1} - 2
   do j = 3, N_{0,2} - 2
      A_2(i, j) = A_1(i - 1, j) + A_1(i, j - 1)+
                  A_1(i, j) + A_1(i, j + 1) + A_1(i + 1, j)
   enddo
enddo
```

Figure 3.11: Exemple de code qui est constitué de deux nids

3.4.2 Fusion

À une itération $\vec{i} \in D_2$ nous calculons $A_2(\vec{i})$ en fonction de 5 données du tableau A_1 qui sont produites dans le nid D_1 par les itérations : $\vec{i_1} = \vec{i} - (1,\ 0)^t$, $\vec{i_2} = \vec{i} - (0,\ 1)^t$, $\vec{i_3} = \vec{i}$, $\vec{i_4} = \vec{i} + (0,\ 1)^t$ et $\vec{i_5} = \vec{i} + (1,\ 0)^t$. Comme le montre la *figure 3.12*, la fusion directe de ces deux nids n'est pas valide, car dans le nid fusionné les itérations $\vec{i_4}$ et $\vec{i_5}$ seront exécutées après l'itération $\vec{i}$. Ce qui introduirait deux *anti-dépendances* de l'instruction du deuxième nid vers l'instruction du premier nid.

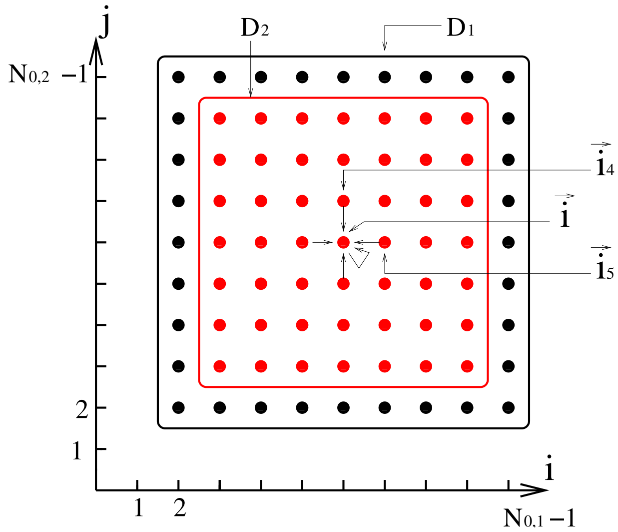


Figure 3.12: Domaines d'itération des deux nids du code en *figure 3.11*

Pour pallier le problème précédent, nous translatons le domaine d'itération du premier nid avec un délai $\vec{h_1} = -(1,\ 0)^t$. Cela revient à avancer tous les calculs du premier nid avec le délai $\vec{h_1}$:

$$
\begin{aligned}
D_1' &= \{\vec{i} = (i,\ j)^t = \vec{i_1} + \vec{h_1}\ : (2,\ 2)^t \le \vec{i_1} \le (N_{0,1} - 1,\ N_{0,2} - 1)^t\} \\
D_1' &= \{\vec{i} = (i,\ j)^t = \vec{i_1} + \vec{h_1}\ : (2,\ 2)^t + \vec{h_1} \le \vec{i} \le (N_{0,1} - 1,\ N_{0,2} - 1)^t + \vec{h_1}\} \\
D_1' &= \{\vec{i} = (i,\ j)^t :\ \ (1,\ 2)^t \le \vec{i} \le (N_{0,1} - 2,\ N_{0,2} - 1)^t\}
\end{aligned}
$$

Le domaine d'itération du deuxième nid est inchangé :

$$D'_2 = D_2 = \{\vec{i} = (i,\ j)^t :\quad (3,\ 3)^t \leq \vec{i} \leq (N_{0,1} - 2,\ N_{0,2} - 2)^t\}$$

Comme le montre la *figure 3.13*, après la translation du domaine d'itération du premier nid, toutes les itérations qui produisent les données qui sont nécessaires pour le calcul d'un élément du tableau A_2 à l'itération $\vec{i}$ seront exécutées avant $\vec{i}$.

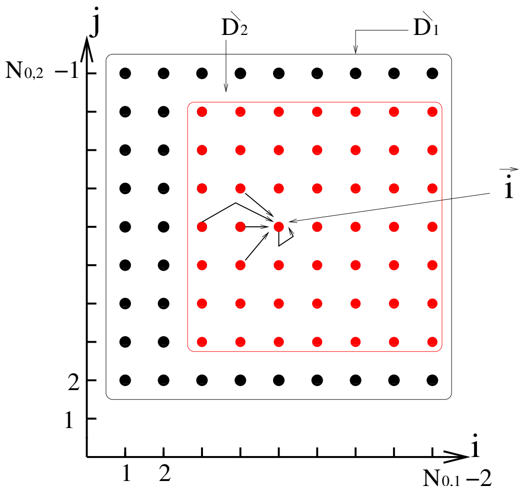


Figure 3.13: Translation du domaine d'itération du premier nid du code en *figure 3.11*

Le code après la translation du domaine D_1 est donné en *figure 3.14*.

```
do i = 1, N_{0,1} − 2
   do j = 2, N_{0,2} − 1
      A_1(i + 1, j) = A_0(i, j) + A_0(i + 1, j − 1)+
                      A_0(i + 1, j) + A_0(i + 1, j + 1) + A_0(i + 2, j)
   enddo
enddo
do i = 3, N_{0,1} − 2
   do j = 3, N_{0,2} − 2
      A_2(i, j) = A_1(i − 1, j) + A_1(i, j − 1)+
                  A_1(i, j) + A_1(i, j + 1) + A_1(i + 1, j)
   enddo
enddo
```

Figure 3.14: Code après la translation du domaine d'itération du premier nid

Le code fusionné après la translation du domaine D_1 est donné en *figure 3.15*.

3.4.3 Fusion avec allocation des tampons

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les mêmes données du tableau A_1, nous remplaçons ce dernier par un tampon séquentiel et circulaire B_1. La taille de ce tampon qui est notée par T_1, sera égale à la borne supérieure Sup_2 du volume mémoire des données vivantes du tableau A_1.

```
do (i = 1, N_{0,1} - 2)
  do (j = 2, N_{0,2} - 1)
     A_1(i + 1, j) = A_0(i, j) + A_0(i + 1, j - 1)+
                  A_0(i + 1, j) + A_0(i, j + 1) + A_0(i + 2, j)
     if ((i >= 3) and (3 <= j <= N_{0,2} - 2))
       A_2(i, j) = A_1(i - 1, j) + A_1(i, j - 1)+
                 A_1(i, j) + A_1(i, j + 1) + A_1(i + 1, j)
  enddo
enddo
```

Figure 3.15: Fusion du code de la *figure 3.11*

Pour calculer Sup_2, nous considérons une itération $\vec{i} \in D'_2$. À cette itération nous utilisons 5 données du tableau A_1 qui sont produites par $\vec{i}_1 = \vec{i} - (2,\ 0)^t$, $\vec{i}_2 = \vec{i} - (1,\ 1)^t$, $\vec{i}_3 = \vec{i} - (1,\ 0)^t$, $\vec{i}_4 = \vec{i} - (1,\ -1)^t$ et $\vec{i}_5 = \vec{i} - (0,\ 0)^t$.

La plus ancienne de ces productions est $\vec{i}_1$. Par conséquent, la borne supérieure Sup_2 du volume mémoire $M_2(\vec{i})$ est donnée par le nombre d'itérations entre $\vec{i}_1$ et $\vec{i}$:

$$\begin{aligned} \mathrm{Sup}_2 &= \vec{C}_1 \ .\ (\vec{i} - \vec{i}_1) + 1 \\ &= (N_{0,2} - 1 - 2 + 1,\ 1)^t \ .(2,\ 0)^t + 1 \\ &= 2 * N_{0,2} - 3 \end{aligned}$$

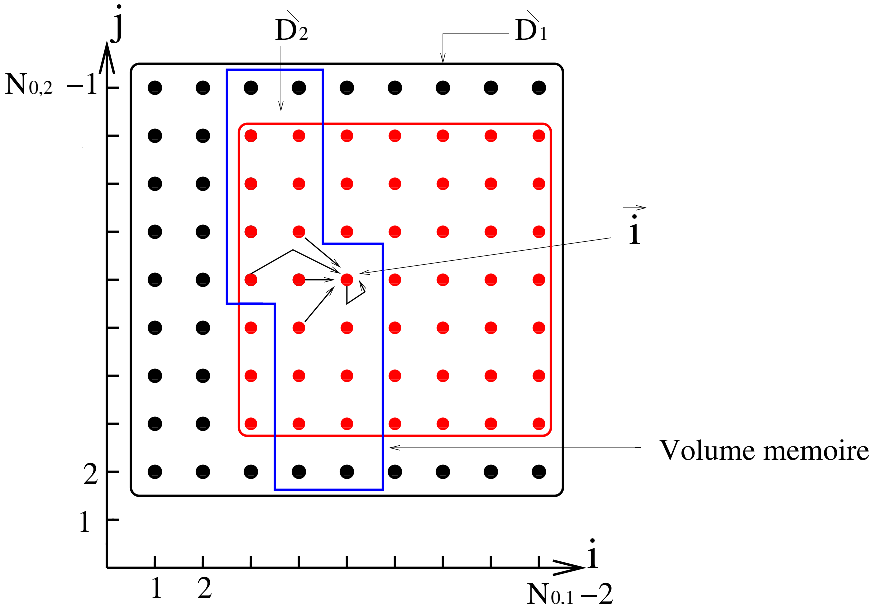


Figure 3.16: Volume mémoire pour le code de la *figure 3.15*

La *figure 3.16* donne les données vivantes du tableau A_1 à une itération $\vec{i} \in D'_2$. Dans cette figure $N_{0,2} = 11$. Par conséquent $\mathrm{Sup}_2 = (9,\ 1)^t \ .\ (2,\ 0)^t + 1 = 19$.

La fonction d'accès F_1 qui sera associée au tampon B_1 est définie comme suit :

$$\begin{aligned} D'_1 &\longrightarrow N \\ F_1 : \vec{i} &\longmapsto [(\vec{i} - (1,2)^t)\ .(N_{0,2} - 2,\ 1)^t] \quad \mathrm{mod}\ T_1 \end{aligned}$$

Après le remplacement du tableau A_1 par le tampon B_1 de taille $T_1 = \mathrm{Sup}_2$, nous obtenons le code donné en *figure 3.17*.

```
do (i = 1, N_{0,1} - 2)
  do (j = 2, N_{0,2} - 1)
    B_1([(i - 1)(N_{0,2} - 2) + (j - 2)] mod T_1)  =  A_0(i, j) + A_0(i + 1, j - 1)+
                                                      A_0(i + 1, j) + A_0(i, j + 1)+
                                                      A_0(i + 2, j)
    if ((i >= 3) and (3 <= j <= N_{0,2} - 2)
      A_2(i, j) = B_1([(i - 3)(N_{0,2} - 2) + (j - 2)] mod T_1)+
                  B_1([(i - 2)(N_{0,2} - 2) + (j - 3)] mod T_1)+
                  B_1([(i - 2)(N_{0,2} - 2) + (j - 2)] mod T_1)+
                  B_1([(i - 2)(N_{0,2} - 2) + (j - 1)] mod T_1)+
                  B_1([(i - 1)(N_{0,2} - 2) + (j - 2)] mod T_1)
  enddo
enddo
```

Figure 3.17: Fusion du code de la *figure 3.5* avec allocation d'un tampon B_1

Les *modulos* dans le code de la *figure 3.17* peuvent être éliminés soit en introduisant des tests supplémentaires soit en déroulant les boucles i et j.

3.5 Conclusion

Le problème de la fusion de boucles a attiré l'attention de nombreux chercheurs depuis plusieurs années [49, 65, 87, 80, 55, 24]. Plusieurs travaux ont mentionné que la fusion de boucles dans le cas général est un problème *NP_ complet* [57, 23, 26]. Par conséquent, plusieurs heuristiques ont été proposées pour résoudre ce problème. Par exemple, Kennedy [57] a proposé un algorithme dans le cas général qui utilise une stratégie gloutonne. Gao et al [32] ont proposé une heuristique dans un cas particulier et leur approche vise l'augmentation de la localité des données. Megido et Sarkar [58] ont utilisé la programmation entière qui donne des solutions optimales mais qui ne sont pas efficaces pour la réutilisation des caches. Pugh et Rosser[62] ont proposé une technique de réordonnancement qui peut produire des résultats similaires à la fusion de boucles. Cette technique est destinée à améliorer la localité des données. Dans ce chapitre, nous nous sommes intéressés à des codes composés de nids de boucles parallèles à domaines constants ; c'est pour cette raison que notre méthode est exacte.

Nous avons montré comment nous pouvons appliquer une fusion sur des codes particuliers qui sont des séquences de nids de boucles qui n'ont pas nécessairement le même domaine d'itération [11, 8, 9, 7, 10]. Notre méthode consiste à décaler les domaines d'itérations des différents nids afin de garantir que toutes les itérations qui produisent des données seront exécutées avant les itérations qui les consomment. Grâce à ces décalages, l'application de notre fusion est toujours possible.

À notre connaissance, la notion de données vivantes n'a pas été utilisée dans le cas de la fusion pour remplacer les différents tableaux intermédiaires. Par contre, la contraction de tableau [54, 30, 32, 67, 25, 70] a été utilisée dans plusieurs travaux dans le cas de réordonnancement des instructions [28, 29, 52]. Dans ce chapitre, nous avons utilisé cette notion de donnée vivante pour remplacer les différents tableaux par des tampons circulaires et séquentiels de taille inférieure. Pour ces différents tampons, nous avons défini des fonctions d'accès qui permettent de stocker et de récupérer les données du tableau correspondant. Les deux techniques présentées dans ce chapitre sont implantées dans $PIPS$ et nous verrons dans le *chapitre 7* que l'allocation des tampons permet de réduire d'une manière considérable le nombre de défauts de cache.

Chapitre 4

Fusion avec pavage à un seul niveau d'une séquence de nids de boucles

Dans la littérature, plusieurs auteurs se sont intéressés au pavage [75, 85, 43, 77], à la fusion de boucles [65, 87, 56], à la fusion de boucles avec alignement [23, 24, 69] et à la permutation de boucles [4, 5]. Dans le cas du pavage, la plupart des travaux sont consacrés à des codes qui sont constitués d'un seul nid de boucles. Dans ce chapitre, nous montrons comment nous combinons toutes ces techniques afin de les appliquer à une séquence de nids de boucles.

Notre pavage est à un seul niveau, c'est-à-dire que nous transformons tous les nids de boucles de profondeur n en un seul nid de profondeur $2n$. Cela revient à partitionner l'espace d'itération en un ensemble de blocs. Les n boucles externes du nid transformé vont parcourir les blocs et les n boucles internes vont parcourir les itérations à l'intérieur d'un bloc.

Irigoin et Triolet [39] ont modélisé le pavage par une matrice $H \in Q^{n*n}$. Les n lignes de cette matrice représentent les vecteurs normaux aux faces du pavage. Ces deux auteurs ont donné la condition suffisante pour l'application de ce pavage. Par contre Xue [83] a proposé d'utiliser le pavage comme une transformation de boucles et a donné la condition nécessaire et suffisante à son application. Dans notre approche, nous avons combiné la modélisation de Xue avec les différentes techniques de fusion de boucles avec alignement [23, 24, 62]. Contrairement à Xue, qui s'est intéressé à des codes qui sont constitués d'un seul nid de boucles, nous traitons des codes qui sont des séquences de nids de boucles.

Les codes en entrée ont la même forme que ceux traités dans le cas de la fusion au *chapitre 3* (*page 10*), c'est-à-dire, des séquences de nids de boucles parallèles ; chacun de ces nids accède à des données suivant un *patron*. Les données utilisées dans un nid sont produites dans le nid précédent. Après l'application du pavage avec fusion, nous devons garantir que toutes les données qui sont nécessaires pour le calcul d'une itération ont déjà été calculées par les itérations précédentes. Pour cela, nous ajoutons un délai [62, 23, 24] à chaque nid qui sera calculé en fonction du pavage et du *patron*.

Contrairement aux autres travaux qui déterminent quel pavage est valide en fonction des dépendances du code initial, nous déterminons quels sont les délais que nous devons ajouter aux nids (modifier les dépendances) pour qu'un pavage donné soit valide. Par conséquent, nous n'essayons pas de déterminer les meilleures tailles de tuiles ; ces dernières seront données par un utilisateur ou déterminées par une série de tests, comme c'est généralement le cas pour du code embarqué de traitement du signal.

Notre pavage sera représenté par deux matrices : (1) une matrice $\mathcal{A}(n,\ 2n)$ de pavage à un seul niveau qui donne les différents coefficients des tuiles et (2) une matrice de permutation $\mathcal{P}(2n,\ 2n)$ qui permet d'échanger plusieurs boucles et donc de spécifier l'ordonnancement des tuiles et des itérations à l'intérieur de ces tuiles. Grâce à la matrice $\mathcal{P}$, nous prenons en considération tous les ordonnancements possibles parallèles aux axes.

De la même manière que pour la fusion, afin de maintenir en mémoire les données vivantes et d'éviter de charger plusieurs fois les mêmes données, nous proposons deux techniques d'allocation de tampons : (1) remplacer chaque tableau par un tampon séquentiel et circulaire et (2) remplacer chaque tableau par un ensemble de sous-tampons.

4.1 Code en entrée

Les codes en entrée ont la même forme que ceux de la fusion, c'est-à-dire qu'ils ont la forme du code en *figure 4.1*

do $\vec{i}_1 \in D_1$
  $A_1(\vec{i}_1) = A_0(\vec{i}_1 + \vec{v}_1^1) \otimes .... \otimes A_0(\vec{i}_1 + \vec{v}_{m_1}^1)$ (* s_1 *)
enddo
.
.
do $\vec{i}_k \in D_k$
  $A_k(\vec{i}_k) = A_{k-1}(\vec{i}_k + \vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i}_k + \vec{v}_{m_k}^k)$ (* s_k *)
enddo
.
.
do $\vec{i}_p \in D_p$
  $A_p(\vec{i}_p) = A_{p-1}(\vec{i}_p + \vec{v}_1^p) \otimes .... \otimes A_{p-1}(\vec{i}_p + \vec{v}_{m_p}^p)$ (* s_p *)
enddo

Figure 4.1: Code en entrée (séquence de nids de boucles). $\forall j : \vec{i}_j \in \mathbb{Z}^n$, la dimension des tableaux A_k est n et $\otimes$ représente une opération quelconque qui dépend de k

Les contraintes sur ces codes sont les mêmes que celles de la fusion, c'est-à-dire que :

1. tous les nids ont la même profondeur, qui est quelconque ;
2. les différents tableaux ont la même dimension, égale à la profondeur des nids ;
3. les bornes de ces différents nids sont des constantes numériques ;
4. les vecteurs des différents patrons sont ordonnés de la manière suivante :
$$\vec{v}_1^k \preceq \vec{v}_2^k \preceq ..... \preceq \vec{v}_{m_k}^k;$$
5. le domaine D_0 est imposé par l'utilisateur et les autres domaines sont dérivés de la manière suivante :
$$D_k = \{\vec{i} \mid \forall \vec{v} \in R^k :\ \vec{i} + \vec{v} \in D_{k-1}\}.$$

Comme nous l'avons mentionné dans la fusion, cette dérivation permet d'éviter d'utiliser des éléments qui ne sont pas définis.

4.2 Pavage avec fusion

Dans le *chapitre 3*, nous avons appliqué uniquement la fusion avec décalage sur une séquence de nids de boucles. Ici, nous appliquons le pavage et la fusion simultanément. Comme nous nous intéressons uniquement à des données vivantes au niveau du cache, notre pavage sera à un seul niveau. Nous transformons donc chaque itération $\vec{i} \in \mathbb{Z}^n$ en une autre itération $\vec{i'} \in \mathbb{Z}^{2n}$. Notre pavage est représenté par deux matrices : (1) une matrice $\mathcal{A}(n,\ 2n)$ de pavage qui permet de spécifier la taille des tuiles et (2) une matrice $\mathcal{P}(2n,\ 2n)$ qui permet d'échanger plusieurs boucles en même temps et donc de spécifier l'ordonnancement des tuiles et des itérations à l'intérieur de ces tuiles. Contrairement aux autres travaux, nous pouvons toujours appliquer notre pavage. Ceci est possible grâce à la translation de chaque domaine d'itération D_k avec un délai $\vec{h}_k$. Ces différents délais seront calculés en fonction des matrices $\mathcal{A}$ et $\mathcal{P}$ et des vecteurs des patrons associés aux différents domaines.

4.2.1 Translation des domaines des différents nids

L'application directe du pavage avec fusion sur le code de la *figure 4.1* n'est pas toujours valide. Pour cela, nous translatons chaque domaine D_k d'un vecteur $\vec{h}_k$ (pour avancer ou retarder les calculs de chaque nid) ; cela revient à modifier les *dépendances de flot* qui existent entre les différents nids sans introduire des *anti-dépendances*. Dans la *section 4.2.7* (*page 40*), nous donnons la relation que doivent vérifier ces délais pour que l'application du pavage avec fusion devienne valide. Après la translation des différents domaines par des délais $\vec{h}_k$, nous obtenons le code en *figure 4.2*.

do $\vec{i} \in D'_1$
$\quad A_1(\vec{i} - \vec{h}_1) = A_0(\vec{i} - \vec{h}_1 + \vec{v}^1_1) \otimes .... \otimes A_0(\vec{i} - \vec{h}_1 + \vec{v}^1_{m_1})$ (* s_1 *)
enddo
.
.
do $\vec{i} \in D'_k$
$\quad A_k(\vec{i} - \vec{h}_k) = A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_1) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_{m_k})$ (* s_k *)
enddo
.
.
do $\vec{i} \in D'_p$
$\quad A_p(\vec{i}_p - \vec{h}_p) = A_{p-1}(\vec{i}_p - \vec{h}_p + \vec{v}^p_1) \otimes .... \otimes A_{p-1}(\vec{i}_p - \vec{h}_p + \vec{v}^p_{m_p})$ (* s_p *)
enddo

Figure 4.2: Séquence de nids de boucles après la translation des différents nids

Soient :

1. $D' = \cup_{k=1}^{p} D'_k$;
2. $\vec{O} = \min_<(\vec{i} \in D')$;
3. $\vec{N} = \max_<(\vec{i} \in D')$ (les opérateurs $\min_<$ et $\max_<$ sont définis dans la *page 7*).

Les vecteurs $\vec{O}$ et $\vec{N}$ ne sont pas nécessairement dans le domaine D' ; nous prenons la plus petite enveloppe convexe qui contiendra $\vec{O}$, $\vec{N}$ et D'. Pour faciliter la génération du code, après le calcul des vecteurs de translation $\vec{h}_k$ qui rendront l'application du pavage avec fusion valide, nous translatons le domaine D' avec un vecteur $\vec{h}$ de façon à assurer que $\vec{O} = \vec{0}$. Cette dernière étape ne modifie pas la condition de légalité du pavage avec fusion, car tous les domaines seront translatés avec le même vecteur.

Définition 14 (Production) *Une itération $\vec{i} \in D'_k$ produit :*

$$A_k(\vec{i} - \vec{h}_k)$$

Définition 15 (Consommation) *Une itération $\vec{i} \in D'_k$ consomme :*

$$\{\ A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_q)\ :\ 1 \leq q \leq m_k)\ \}$$

Définition 16 *Une itération $\vec{i} \in D'_{k+1}$ consomme l'élément produit par $\vec{j} \in D'_k$ si et seulement si :*

$$\exists \vec{v} \in R^{k+1}\ :\ \vec{j} = \vec{i} - \vec{h}_{k+1} + \vec{h}_k + \vec{v}$$

La *définition 16* signifie qu'il existe une *dépendance de flot* entre $\vec{j}$ et $\vec{i}$.

4.2.2 Matrice du pavage A

La matrice du pavage $\mathcal{A}(n,\ 2n)$ permet de spécifier les tailles des tuiles. Cette matrice a la forme suivante :

$$\begin{pmatrix}
a_{1,1} & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & a_{i,2i-1} & 1 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & a_{n,2n-1} & 1
\end{pmatrix}$$

Tous les éléments de la $i^{\text{ème}}$ ligne de cette matrice sont nuls sauf :

1. $a_{i,2i-1}$ qui représente la taille des tuiles sur le $i^{\text{ème}}$ axe ;
2. $a_{i,2i}$ qui est égal à 1.

À partir de la matrice $\mathcal{A}$, nous pouvons définir une transformation ω_1 qui permet de transformer chaque point $\vec{i} = (i_1, \cdots,\ i_n)^t \in D' \subset \mathbb{Z}^n$ (*figure 4.2*) en un autre point $\vec{i'} = (i'_1, \cdots,\ i'_{2n})^t \in \omega_1(D') \subset \mathbb{Z}^{2n}$ (*figure 4.3*) :

$$\begin{aligned}
D' \quad &\longrightarrow \quad \omega_1(D') \subset \mathbb{Z}^{2n} \\
\omega_1 : \vec{i} \quad &\longmapsto \quad \vec{i'} = (\lfloor \frac{i_1}{a_{1,1}} \rfloor,\ \ i_1 \bmod\ a_{1,1}, \cdots,\ \lfloor \frac{i_m}{a_{m,2m-1}} \rfloor,\ \ i_m \bmod\ a_{m,2m-1}, \cdots, \\
&\qquad\qquad \lfloor \frac{i_n}{a_{n,2n-1}} \rfloor,\ \ i_n \bmod\ a_{n,2n-1})^t.
\end{aligned}$$

Sachant que chaque composante O_i du vecteur $\vec{O} = \min_{<}(\vec{i} \in D')$ est un multiple de $a_{i,2i-1}$. Afin de faciliter la génération du code, tout au long de ce chapitre nous supposons que $\vec{O} = \vec{0}$. Nous verrons dans la *section 4.2.9 (page 45)*, comment nous pouvons satisfaire cette condition.

Définition 17 *Soit $\vec{T}$ le vecteur des tailles des tuiles : $T_i = a_{i,2i-1}$. À partir du vecteur $\vec{i'}$, nous pouvons extraire deux sous vecteurs :*

1. *un vecteur $\vec{i'_{\mathcal{E}}} = \lfloor \frac{\vec{i}}{\vec{T}} \rfloor = (\lfloor \frac{i_1}{a_{1,1}} \rfloor, \cdots, \lfloor \frac{i_m}{a_{m,2m-1}} \rfloor, \cdots, \lfloor \frac{i_n}{a_{n,2n-1}} \rfloor)^t$ qui parcourt les tuiles.*
2. *un vecteur $\vec{i'_{\mathcal{R}}} = \vec{i} \bmod \vec{T} = (i_1 \bmod a_{1,1}, \cdots, i_m \bmod a_{m,2*m-1}, \cdots, i_n \bmod a_{n,2*n-1})^t$ qui parcourt les itérations à l'intérieur des tuiles.*

Propriété 1 *Nous pouvons écrire $\vec{i'}$ en fonction de $\vec{i'_{\mathcal{E}}}$ et $\vec{i'_{\mathcal{R}}}$ de la manière suivante :*

$$\vec{i'} = \mathcal{M} \begin{pmatrix} \vec{i'_{\mathcal{E}}} \\ \vec{i'_{\mathcal{R}}} \end{pmatrix},$$

$\mathcal{M}$ est une matrice de taille $(2n,\ 2n)$ qui sera notée par $\big(\mathcal{M}_1 \mathcal{M}_2\big)$, avec :

$$\mathcal{M}_1(2n,\ n) = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } \mathcal{M}_2(2n,\ n) = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

□

La transformation ω_1 est bijective [84] et nous avons $\vec{i} = \mathcal{A}\ \vec{i'}$. Après l'application de la matrice $\mathcal{A}$ sur le code en *figure 4.2* nous obtenons le code en *figure 4.3.*

```
do(i' ∈ ω1(D'))
   C1(i') :  A1(Ai' − h1) = A0(Ai' − h1 + v1^1) ⊗ ··· ⊗ A0(Ai' − h1 + v^1_m1)        (* s1 *)
   .
   .
   Ck(i') :  Ak(Ai' − hk) = Ak−1(Ai' − hk + v1^k) ⊗ ··· ⊗ Ak−1(Ai' − hk + v^k_mk)    (* sk *)
   .
   .
   Cp(i') Ap(Ai' − hp) = Ap−1(Ai' − hp + v1^p) ⊗ ··· ⊗ Ap−1(Ai' − hp + v^p_mp)       (* sn *)
enddo
```

Figure 4.3: Code après application de $\mathcal{A}$ et translation du domaine d'itération de chaque nid avec un délai $\vec{h}_k$

Comme l'instruction S_k n'est pas exécutée pour toute itération du domaine $\omega_1(D')$ du code en *figure 4.3*, elle doit être gardée par la condition :

$$C_k(\vec{i'}) = \text{if } (\vec{i'} \in \omega_1(D'_k))$$

Dans le code tuilé, les boucles qui parcourent les tuiles doivent être placées avant les boucles qui parcourent les itérations dans les tuiles. Dans le code en *figure 4.3*, cette condition n'est pas vérifiée. Par conséquent, nous appliquons une matrice de permutation $\mathcal{P}$ sur ce code afin d'échanger plusieurs boucles en même temps.

4.2.3 Échange de boucles

La matrice de pavage $\mathcal{A}(n,\ 2n)$ permet uniquement de spécifier la taille des tuiles. Elle n'a par contre aucune influence sur l'ordre d'exécution des itérations par rapport au code initial. Grâce à la matrice de permutation $\mathcal{P}(2n,\ 2n)$ qui permet d'échanger plusieurs boucles en même temps, nous pouvons spécifier l'ordre dans lequel les tuiles et les itérations à l'intérieur de ces tuiles seront exécutées. Cette matrice aura la forme suivante :

$$\begin{pmatrix} p_{1,1} & \cdots & p_{1,j} & \cdots & p_{1,2n} \\ \vdots & \ddots & \vdots & & \vdots \\ p_{i,1} & \cdots & p_{i,j} & \cdots & p_{i,2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{2n,1} & \cdots & p_{2n,j} & \cdots & p_{2n,2n} \end{pmatrix}$$

Par définition d'une matrice de permutation, dans chaque ligne et dans chaque colonne de la matrice $\mathcal{P}$, il y a un et un seul élément qui est égal à 1 et les autres éléments sont tous nuls. C'est-à-dire que :

$$\begin{array}{l} \forall i,\ \exists! j:\ \mathcal{P}_{i,j} = 1 \\ \forall j,\ \exists! i:\ \mathcal{P}_{i,j} = 1 \end{array}$$

La matrice $\mathcal{P}$ transforme chaque point $\vec{i'} \in \mathbb{Z}^{2n}$ du code en *figure 4.3* en un autre point $\vec{l} \in \mathbb{Z}^{2n}$ du code en *figure 4.4*.

do $[\vec{l} \in \mathcal{P}\ (\omega_1(D'))\ ]$

$C_1(\vec{l}):\ A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$ (* s_1 *)

.

.

$C_k(\vec{l}):\ A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)$ (* s_k *)

.

.

$C_p(\vec{l}):\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)$ (* s_p *)

enddo

Figure 4.4: Code après application de la matrice $\mathcal{P}$

L'instruction S_k doit être gardée par :

$$C_k(\vec{l}) = \text{if } [\ \vec{l} \in \mathcal{P}\ (\omega_1(D'_k))\ ]$$

La relation entre $\vec{i'}$ et $\vec{l}$ est donnée par :

$$\begin{array}{rcl} \vec{l} & = & \mathcal{P}\ \vec{i'} \\ \vec{i'} & = & \mathcal{P}^{-1}\ \vec{l} \end{array}$$

4.2.4 Pavage contigu

Lorsque les itérations d'une même tuile sont exécutées d'une manière séquentielle, le pavage sera dit contigu. La *définition 18*, donne la définition formelle d'un pavage contigu.

Définition 18 (Pavage contigu) *Un pavage est dit contigu si et seulement si :*

1. $\forall k \in [1 \cdots n]$, $\exists j$: $l_k = \lfloor \frac{i_j}{a_{j,2j-1}} \rfloor$
2. $\forall k \in [n+1 \cdots 2n]$, $\exists j$: $l_k = (i_j \bmod a_{j,2j-1})$

La *définition 18* signifie que toutes les boucles qui parcourent les tuiles $(\vec{i'_{\mathcal{E}}})$ doivent être placées avant les boucles qui parcourent les itérations à l'intérieur de ces tuiles $(\vec{i'_{\mathcal{R}}})$.

Certaines configurations de la matrice $\mathcal{P}$ ne sont pas admises, car elles ne donnent pas un pavage contigu. La *proposition 6* donne les conditions que doit vérifier la matrice $\mathcal{P}$ pour que notre pavage soit contigu.

Proposition 6 *Un pavage est contigu si et seulement si :*

$$\begin{cases} \forall k \ : \ [(\exists j \ : \ \mathcal{P}_{k,2j-1} = 1) \Rightarrow k \leq n] \\ \forall k \ : \ [(\exists j \ : \ \mathcal{P}_{k,2j} = 1) \Rightarrow k \geq n+1] \end{cases}$$

Preuve :

$\Rightarrow$? Nous supposons que :
- (a) notre pavage est contigu et
- (b) $\exists k \ : \ [\ (\exists j \ : \ \mathcal{P}_{k,2j-1} = 1) \wedge (k \geq n+1)]$ ou $\exists k \ : \ [\ (\exists j \ : \ \mathcal{P}_{k,2j} = 1) \wedge (k \leq n)]$

1) $(\mathcal{P}_{k,2j-1} = 1) \wedge (k \geq n+1) \ \Rightarrow \ l_k = \lfloor \frac{i_j}{a_{j,2j-1}} \rfloor \wedge (k \geq n+1)$

$\Rightarrow$ Contradiction avec l'hypothèse (a)(*définition 18*).

2) $(\mathcal{P}_{k,2j} = 1) \wedge (k \leq n) \ \Rightarrow \ l_k = (i_j \bmod a_{j,2j-1}) \wedge (k \leq n)$

$\Rightarrow$ Contradiction avec l'hypothèse (a) (*définition 18*).

$\Leftarrow$? Nous supposons que :
- (a) $\begin{cases} \forall k \ : [\ (\exists j \ : \ \mathcal{P}_{k,2j-1} = 1) \Rightarrow k \leq n] \\ \forall k \ : [\ (\exists j \ : \ \mathcal{P}_{k,2j} = 1) \Rightarrow k \geq n+1] \end{cases}$
- (b) notre pavage n'est pas contigu.

  D'après la *définition 18*, le pavage peut ne pas être contigu dans deux cas :

  Cas1 : $\exists k \in [1 \cdots n], \forall j \ : l_k \neq \lfloor \frac{i_j}{a_{j,2j-1}} \rfloor$. Cela signifie que :

$$\exists k \in [1 \cdots n], \exists j \ : l_k = (i_j \bmod a_{j,2j-1})$$

  Par conséquent :

$$\exists k \in [1 \cdots n], \exists j : \mathcal{P}_{k,2j} = 1$$

  Contradiction avec l'hypothèse (a).

  Cas 2 : $\exists k \in [n+1 \cdots 2n], \forall j \ : l_k \neq (i_j \bmod a_{j,2j-1})$. Cela signifie que :

$$\exists k \in [n+1 \cdots 2n], \exists j \ : \ l_k = \lfloor \frac{i_j}{a_{j,2j-1}} \rfloor$$

Par conséquent :

$$\exists k \in [n+1 \cdots 2n], \exists j \ : \mathcal{P}_{a,2j-1} = 1$$

Contradiction avec l'hypothèse (a).

$\square$

Tout au long de ce chapitre, nous ne considérons que des matrices $\mathcal{P}(2n,\ 2n)$ qui donnent un pavage contigu.

4.2.5 Décomposition de la matrice $\mathcal{P}$

Définition 19 *À partir de la matrice $\mathcal{P}(2n,\ 2n)$, nous pouvons extraire deux sous matrices :*

1. *une matrice $\mathcal{E}(n,\ n)$ de permutation des boucles externes $(\vec{i'}_{\mathcal{E}})$: les boucles qui parcourent les tuiles : $\mathcal{E}_{i,j} = \mathcal{P}_{i,2j-1}$;*
2. *une matrice $\mathcal{R}(n,\ n)$ de permutation des boucles internes $(\vec{i'}_{\mathcal{R}})$: les boucles qui parcourent les itérations à l'intérieur des tuiles : $\mathcal{R}_{i,j} = \mathcal{P}_{i+n,2j}$.*

Proposition 7 *Si la matrice $\mathcal{P}$ satisfait aux conditions données dans la* proposition 6 *alors :*

$$\vec{l} = \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix}$$

Preuve : D'après la *propriété 1* (*page 35*) :

$$\vec{i'} = \mathcal{M} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix}$$

$\mathcal{M}$ est une matrice de taille $(2n,\ 2n)$ que nous pouvons décomposer en deux sous matrices :

$$\mathcal{M}_1(2n,\ n) = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ et } \mathcal{M}_2(2n,\ n) = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Comme $\vec{l} = \mathcal{P}\vec{i'}$ alors $\vec{l} = \mathcal{P} \ \begin{pmatrix} \mathcal{M}_1 & \mathcal{M}_2 \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix}$.

Nous pouvons vérifier facilement que $\mathcal{P} \ \begin{pmatrix} \mathcal{M}_1 & \mathcal{M}_2 \end{pmatrix} = \begin{pmatrix} \mathcal{Q}_{1,1} & \mathcal{Q}_{1,2} \\ \mathcal{Q}_{2,1} & \mathcal{Q}_{2,2} \end{pmatrix}$ Avec :

$$\mathcal{Q}_{1,1} = \begin{pmatrix} p_{1,1} & p_{1,3} & \cdots & p_{1,2n-1} \\ \vdots & \vdots & & \vdots \\ p_{i,1} & p_{i,3} & \cdots & p_{i,2n-1} \\ \vdots & \vdots & & \vdots \\ p_{n,1} & p_{n,3} & \cdots & p_{n,2n-1} \end{pmatrix}, \mathcal{Q}_{1,2} = \begin{pmatrix} p_{1,2} & p_{1,4} & \cdots & p_{1,2n} \\ \vdots & \vdots & & \vdots \\ p_{i,2} & p_{i,4} & \cdots & p_{i,2n} \\ \vdots & \vdots & & \vdots \\ p_{n,2} & p_{n,4} & \cdots & p_{n,2n} \end{pmatrix},$$

$$\mathcal{Q}_{2,1} = \begin{pmatrix} p_{n+1,1} & p_{n+1,3} & \cdots & p_{n+1,2n-1} \\ \vdots & \vdots & & \vdots \\ p_{n+i,1} & p_{n+i,3} & \cdots & p_{n+i,2n-1} \\ \vdots & \vdots & & \vdots \\ p_{2n,1} & p_{2n,3} & \cdots & p_{2n,2n-1} \end{pmatrix} \text{ et } \mathcal{Q}_{2,2} = \begin{pmatrix} p_{n+1,2} & p_{n+1,4} & \cdots & p_{n+1,2n} \\ \vdots & \vdots & & \vdots \\ p_{n+i,2} & p_{n+i,4} & \cdots & p_{n+i,2n} \\ \vdots & \vdots & & \vdots \\ p_{2n,2} & p_{2n,4} & \cdots & p_{2n,2n} \end{pmatrix}$$

D'après la *définition 19* (*page 38*) :

$$\mathcal{E} = \mathcal{Q}_{1,1} \text{ et } \mathcal{R} = \mathcal{Q}_{2,2}$$

D'après la *proposition 6* (*page 37*), les éléments des matrices $\mathcal{Q}_{2,1}$ et $\mathcal{Q}_{1,2}$ sont différents de 1. Par conséquent, ils sont nuls (définition d'une matrice de permutation $\mathcal{P}$). Par conséquent :

$$\vec{l} = \begin{pmatrix} \mathcal{Q}_{1,1} & \mathcal{Q}_{1,2} \\ \mathcal{Q}_{2,1} & \mathcal{Q}_{2,2} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix} = \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix}$$

□

4.2.6 Modélisation du pavage

Contrairement à la fusion sans pavage, le calcul des différents délais $\vec{h}_k$ pour le pavage avec fusion du code de la *figure 4.1* nécessite plus d'efforts. Comme nous l'avons signalé précédemment, parmi les auteurs qui se sont intéressés au pavage, le seul qui a modélisé le pavage comme une transformation de boucles est Xue [83]. Mais ce dernier s'est intéressé à un seul nid de boucles et à un seul pavage (la matrice $\mathcal{P}$ est fixe).

Notre pavage avec fusion sera représenté par une transformation de boucles ω_2 qui transforme chaque point $\vec{i} \in D' = \cup_{k=1}^{p} D'_k$ en un autre point $\vec{l} \in \mathbb{Z}^{2n}$.

$$\begin{aligned} D' &\longrightarrow \mathbb{Z}^{2n} \\ \omega_2 : \vec{i} &\longmapsto \vec{l} = \mathcal{P} \, .\vec{i'} \\ &\qquad = \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \end{pmatrix} \end{aligned} \tag{4.1}$$

4.2.7 Condition de légalité du pavage avec fusion

De la même manière que la fusion, pour que notre pavage avec fusion soit valide, nous devons garantir que toutes les données du tableau A_k qui sont nécessaires pour le calcul d'un élément du tableau A_{k+1} à une itération $\vec{l}$ dans le code tuilé et fusionné ont déjà été produites par les itérations précédentes.

Définition 20 (Pavage avec fusion valide) *Le pavage avec fusion défini par ω_2 est valide si et seulement si :*

$$\forall \vec{i} \in D'_{k+1},\ \forall \vec{j} \in D'_k \ : \vec{i} \text{ consomme l'élément produit par } \vec{j} \Rightarrow \omega_2(\vec{j}) \preceq \omega_2(\vec{i})$$

Lemme 6 *Le pavage avec fusion défini par ω_2 est valide si et seulement si :*

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q : \omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \preceq \omega_2(\vec{i}) \tag{4.2}$$

Preuve :

$\Rightarrow$? Nous supposons que :

- (a) notre pavage avec fusion est valide et
- (b) $\exists \vec{i} \in D'_{k+1}, \exists k, \exists q : \omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \succ \omega_2(\vec{i})$.

Soit $\vec{j} = \vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k$. D'après la *définition 16* (*page 34*), $\vec{i}$ consomme l'élément produit par $\vec{j}$. Comme $\omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \succ \omega_2(\vec{i})$ alors :

$$\omega_2(\vec{j}) \succ \omega_2(\vec{i})$$

Ceci est contradictoire avec *l'hypothèse a* (*définition 20*).

$\Leftarrow$? Nous supposons que :

- (a) $\forall \vec{i} \in D'_{k+1}, \forall k, \forall q : \omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \preceq \omega_2(\vec{i})$ et
- (b) notre pavage avec fusion n'est pas valide.

D'après la *définition 20*, notre pavage avec fusion n'est pas valide alors :

$$\exists \vec{i} \in D'_{k+1}, \exists \vec{j} \in D'_k : \vec{i} \text{ consomme l'élément produit par } \vec{j} \text{ et } \omega_2(\vec{j}) \succ \omega_2(\vec{i})$$

D'après la *définition 16* (*page 34*), si $\vec{i}$ consomme l'élément produit par $\vec{j}$ alors :

$$\exists \vec{v}_q^{k+1} : \vec{j} = \vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k$$

Cela signifie que :

$$\omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \succ \omega_2(\vec{i})$$

Ceci est contradictoire avec *l'hypothèse a*.

□

Théorème 3 *Soit* $\vec{d}_q^k = -\vec{v}_q^{k+1} + \vec{h}_{k+1} - \vec{h}_k$ la dépendance de flot *de l'instruction S_k vers l'instruction S_{k+1} et $\vec{T}$ le vecteur des tailles d'une tuile ; $T_i = a_{i,2i-1}$. Le* lemme 6 (page 40) *est équivalent à :*

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q : \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} \preceq \begin{pmatrix} \mathcal{E} & 0 \\ 0 & R \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ (\vec{i}) \bmod \vec{T} \end{pmatrix} \tag{4.3}$$

Preuve : En posant $\vec{j} = \vec{i} - \vec{d}_q^k$, le *lemme 6* (*page 40*) sera équivalent à :

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q \ : \ \omega_2(\vec{j}) \preceq \omega_2(\vec{i})$$

D'après la définition de la transformation ω_2 (*page 39*) :

$$\begin{aligned}
\omega_2(\vec{j}) &= \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \vec{j}'_{\mathcal{E}} \\ \vec{j}'_{\mathcal{R}} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{j}}{\vec{T}} \rfloor \\ \vec{j} \bmod \vec{T} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} \\
\omega_2(\vec{i}) &= \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \vec{i}'_{\mathcal{E}} \\ \vec{i}'_{\mathcal{R}} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix}
\end{aligned}$$

Par conséquent le *lemme 6* (*page 40*) sera équivalent à :

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q : \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} \preceq \begin{pmatrix} \mathcal{E} & 0 \\ 0 & R \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix}$$

□

4.2.8 Solutions possibles

Contrairement aux transformations unimodulaires [5], la validité de l'application du pavage avec fusion ne dépend pas uniquement des dépendances mais aussi du vecteur d'itérations $\vec{i}$ et de la taille des tuiles (*théorème 3*). Ici, nous proposons les deux solutions les plus importantes de validité du pavage avec fusion défini par ω_2, qui sont données respectivement par le *théorème 4* (*page 41*) et le *théorème 6* (*page 43*).

Théorème 4 *Si* $\forall k$, $\forall q$: $\vec{d}_q^k = -\vec{v}_q^{k+1} + \vec{h}_{k+1} - \vec{h}_k \geq \vec{0}$ *alors le pavage avec fusion défini par* ω_2 *est valide.*

Preuve : Comme $\vec{d}_q^k \geq \vec{0}$, alors nous avons deux cas :

1. Cas 1 : $\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \leq \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$ et $\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \neq \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$. Comme $\mathcal{E}$ est une matrice de permutation alors :

$$\mathcal{E} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}} \rfloor \leq \mathcal{E} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \text{ et } \mathcal{E} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}} \rfloor \neq \mathcal{E} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$$

   Nous savons que si $\vec{a} \leq \vec{b}$ et $\vec{a} \neq \vec{b}$ alors $\vec{a} \prec \vec{b}$. Par conséquent :

$$\mathcal{E} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}} \rfloor \prec \mathcal{E} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$$

   Comme nous avons :

   (a) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor) \\ \mathcal{R}[(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \end{pmatrix}$

(b) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor \frac{\vec{i}}{\vec{T}} \rfloor) \\ \mathcal{R}[\vec{i} \bmod \vec{T}] \end{pmatrix}$

(c) $\mathcal{E} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \prec \mathcal{E} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$

Alors :

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q \ : \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} \prec \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix}$$

D'après le *Théorème 3* (*page 40*), le pavage avec fusion défini par ω_2 est valide.

2. Cas 2 : $\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor = \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$.

   Nous savons que $\vec{a} \bmod \vec{b} = \vec{a} - \vec{b}\ o\ \lfloor \frac{\vec{a}}{\vec{b}} \rfloor$ (*définition 9* dans la *page 8*). Par conséquent :

   (a) $(\vec{i} - \vec{d}_q^k) \bmod \vec{T} = (\vec{i} - \vec{d}_q^k) - \vec{T}\ o\ \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor$

   (b) $\vec{i} \bmod \vec{T} = \vec{i} - \vec{T}\ o\ \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$

   Comme $(\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor = \lfloor \frac{\vec{i}}{\vec{T}} \rfloor)$ et $(\vec{d}_q^k \geq \vec{0})$ alors $(\vec{i} - \vec{d}_q^k) - \vec{T}\ o\ \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \leq \vec{i} - \vec{T} \mathrm{o}\ \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$. Cela signifie que :

$$(\vec{i} - \vec{d}_q^k) \bmod \vec{T} \leq \vec{i} \bmod \vec{T}$$

   Comme $\mathcal{R}$ est une matrice de permutation alors $\mathcal{R}\ [(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \leq \mathcal{R}\ [\vec{i} \bmod \vec{T}]$. D'après la définition de l'opérateur $\preceq$ (*définition 6* dans la *page 7*), nous déduisons que

$$\mathcal{R}\ [(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \preceq \mathcal{R}\ [\vec{i} \bmod \vec{T}]$$

   Comme nous avons :

   (a) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor) \\ \mathcal{R}[(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \end{pmatrix}$

   (b) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor \frac{\vec{i}}{\vec{T}} \rfloor) \\ \mathcal{R}[\vec{i} \bmod \vec{T}] \end{pmatrix}$

   (c) $\mathcal{E} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor = \mathcal{E} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor$

   (d) $\mathcal{R}\ [(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \preceq \mathcal{R}\ [\vec{i} \bmod \vec{T}]$

   Alors :

$$\forall k, \forall \vec{i} \in D'_{k+1}, \forall q \ : \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} \preceq \begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}} \rfloor \\ (\vec{i}) \bmod \vec{T} \end{pmatrix}$$

   D'après le *théorème 3* (*page 40*), le pavage avec fusion défini par ω_2 est valide.

□

Théorème 5 *Si* $\forall k \ : \ \vec{h}_k = \vec{h}_{k+1} - \max_{<}(\vec{v} \in R^{k+1})$ *alors le pavage avec fusion défini par* ω_2 *est valide.*

Preuve :

$$\vec{h}_k = \vec{h}_{k+1} - \max_{<}(\vec{v} \in R^{k+1}) \Rightarrow \vec{d}_q^k = -\vec{v}_q^{k+1} + \max_{<}(\vec{v} \in R^{k+1})$$
$$\Rightarrow \forall k, \forall q \ : \vec{d}_q^k \geq \vec{0}.$$

D'après le *théorème 4 (page 41)*, le pavage avec fusion défini par ω_2 est valide.

□

Avant de donner la deuxième solution, nous donnons la *proposition 8* que nous utiliserons pour prouver le *théorème 6*.

Proposition 8 *Soient $\mathcal{E}_j$ la ligne j de la matrice $\mathcal{E}$ et $\mathcal{R}_j$ la ligne j de la matrice $\mathcal{R}$. Nous avons les relations suivantes :*

1. $\mathcal{E}_j \ (\vec{a}+\vec{b}) = \mathcal{E}_j \ \vec{a} + \mathcal{E}_j \ \vec{b}$
2. $\mathcal{E}_j \ \lfloor \frac{\vec{a}}{\vec{b}} \rfloor = \lfloor \frac{\mathcal{E}_j \ \vec{a}}{\mathcal{E}_j \ \vec{b}} \rfloor$
3. $\mathcal{R}_j \ (\vec{a}+\vec{b}) = \mathcal{R}_j \ \vec{a} + \mathcal{R}_j \ \vec{b}$
4. $\mathcal{R}_j \ \lfloor \frac{\vec{a}}{\vec{b}} \rfloor = \lfloor \frac{\mathcal{R}_j \ \vec{a}}{\mathcal{R}_j \ \vec{b}} \rfloor$

Preuve : Les preuves du (1) et (3) découlent des propriétés du produit scalaire et celles du (2) et (4) découlent du fait que chaque ligne des matrices $\mathcal{E}$ et $\mathcal{R}$ possède un seul élément qui est égal à 1 et que les autres éléments sont nuls.

□

Théorème 6 *Si pour tout k et pour toute dépendance $\vec{d}_q^k$, $\mathcal{E}\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor \succeq \vec{0}$, alors le pavage avec fusion défini par ω_2 est valide.*

Preuve : D'après la définition de l'opérateur $\succeq$ (*définition 6* dans la *page 7*), $\mathcal{E}\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor \succeq \vec{0}$ donne deux cas :

1. Cas 1 : $\mathcal{E}\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor \geq \vec{0}$.

   Comme $\mathcal{E}\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor = \lfloor \frac{\mathcal{E} \ \vec{d}_q^k}{\mathcal{E} \ \vec{T}} \rfloor \geq \vec{0}$ (*proposition 8*) et $\mathcal{E} \ \vec{T} > 0$ alors :

   $$\forall k, \ \forall q \ : \ \mathcal{E} \ \vec{d}_q^k \geq \vec{0}$$

   Comme $\mathcal{E}$ est une matrice de permutation alors :

   $$\forall k, \ \forall q \ : \ \vec{d}_q^k \geq \vec{0}$$

   D'après le *Théorème 4 (page 41)*, le pavage avec fusion défini par ω_2 est valide.

2. Cas 2 : $\exists i > 1 \ : \ \mathcal{E}_i\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor < 0$.

   Soit i_1 le plus petit des $i > 1$ tel que : $\mathcal{E}_{i_1}\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor < 0$. C'est-à-dire que $\forall j < i_1 \ : \ \mathcal{E}_j\lfloor \frac{\vec{d}_q^k}{\vec{T}} \rfloor \geq 0$.

   Cela signifie que $\forall j < i_1 \ : \ \lfloor \frac{\mathcal{E}_j \vec{d}_q^k}{\mathcal{E}_j \vec{T}} \rfloor \geq 0$ (*proposition 8*). Comme $\forall j < i_1 \ : \ \mathcal{E}_j \vec{T} > 0$ alors :

   $$\forall j < i_1 \ : \ \mathcal{E}_j \ \vec{d}_q^k \geq 0.$$

Par conséquent :

$$\forall j < i_1 \ : \ \mathcal{E}_j\vec{i} - \mathcal{E}_j\vec{d}_q^k \leq \mathcal{E}_j\vec{i}$$

Comme $\forall j < i_1 \ : \ \mathcal{E}_j\vec{T} > 0$ alors :

$$\forall j < i_1 \ : \ \lfloor\frac{\mathcal{E}_j\vec{i} - \mathcal{E}_j\vec{d}_q^k}{\mathcal{E}_j\vec{T}}\rfloor \leq \lfloor\frac{\mathcal{E}_j\vec{i}}{\mathcal{E}_j\vec{T}}\rfloor$$

En utilisant les propriétés de la *proposition 8* (*page 43*), nous obtenons :

$$\forall j < i_1 \ : \ \mathcal{E}_j\lfloor\frac{\vec{i} - \vec{d}_q^k}{\vec{T}}\rfloor \leq \mathcal{E}_j\lfloor\frac{\vec{i}}{\vec{T}}\rfloor.$$

Comme $\mathcal{E}\lfloor\frac{\vec{d}_q^k}{\vec{T}}\rfloor \succeq \vec{0}$ et $\mathcal{E}_{i_1}\lfloor\frac{\vec{d}_q^k}{\vec{T}}\rfloor < 0$ alors :

$$\exists\ i_2 < i_1 \ : \ \mathcal{E}_{i_2}\lfloor\frac{\vec{d}_q^k}{\vec{T}}\rfloor \geq 1$$

En utilisant à nouveau les propriétés de la *proposition 8* (*page 43*), nous montrons que :

$$\lfloor\frac{\mathcal{E}_{i_2}\vec{d}_q^k}{\mathcal{E}_{i_2}\vec{T}}\rfloor \geq 1 \Rightarrow \mathcal{E}_{i_2}\vec{d}_q^k \geq \mathcal{E}_{i_2}\vec{T}$$

Comme $\mathcal{E}_{i_2}\vec{T} > 0$ alors $\mathcal{E}_{i_2}\vec{d}_q^k > 0$. Par conséquent :

$$\lfloor\frac{\mathcal{E}_{i_2}\vec{i} - \mathcal{E}_{i_2}\vec{d}_q^k}{\mathcal{E}_{i_2}\vec{T}}\rfloor < \lfloor\frac{\mathcal{E}_{i_2}\vec{i}}{\mathcal{E}_{i_2}\vec{T}}\rfloor$$

Cela signifie que (en utilisant la *proposition 8*) :

$$\mathcal{E}_{i_2}\lfloor\frac{\vec{i} - \vec{d}_q^k}{\vec{T}}\rfloor < \mathcal{E}_{i_2}\lfloor\frac{\vec{i}}{\vec{T}}\rfloor$$

Nous avons donc :

(a) $\forall j < i_1 \ : \ \mathcal{E}_j\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor \leq \mathcal{E}_j\lfloor\frac{\vec{i}}{\vec{T}}\rfloor$.

(b) $\exists i_2 < i_1 \ : \ \mathcal{E}_{i_2}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor < \mathcal{E}_{i_2}\lfloor\frac{\vec{i}}{\vec{T}}\rfloor$

Qui peuvent être combinées sous la forme :

$$\mathcal{E}\lfloor\frac{\vec{i} - \vec{d}_q^k}{\vec{T}}\rfloor \prec \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}}\rfloor$$

Comme nous avons :

(a) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor) \\ \mathcal{R}[(\vec{i} - \vec{d}_q^k) \bmod \vec{T}] \end{pmatrix}$

(b) $\begin{pmatrix} \mathcal{E} & 0 \\ 0 & \mathcal{R} \end{pmatrix} \begin{pmatrix} \lfloor\frac{\vec{i}}{\vec{T}}\rfloor \\ \vec{i} \bmod \vec{T} \end{pmatrix} = \begin{pmatrix} \mathcal{E}(\lfloor\frac{\vec{i}}{\vec{T}}\rfloor) \\ \mathcal{R}[\vec{i} \bmod \vec{T}] \end{pmatrix}$

(c) $\mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor \prec \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}}\rfloor$

Alors :

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q\ :\ \begin{pmatrix}\mathcal{E} & 0\\ 0 & \mathcal{R}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}}\rfloor\\ (\vec{i}-\vec{d}_q^k) \bmod \vec{T}\end{pmatrix} \prec \begin{pmatrix}\mathcal{E} & 0\\ 0 & \mathcal{R}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}}{\vec{T}}\rfloor\\ \vec{i} \bmod \vec{T}\end{pmatrix}$$

D'après le *théorème 3* (*page 40*), le pavage avec fusion défini par ω_2 est valide.

□

La solution donnée dans ce dernier *théorème* n'est pas intéressante car les projections des dépendances sur certains axes sont très grandes lorsque les tuiles sont larges. Dans la suite de ce chapitre, nous considérons la solution donnée dans le *théorème 4* (*page 41*).

4.2.9 Les étapes de calcul des délais $\vec{h}_k$

Comme nous pouvons le remarquer dans le *théorème 4* (*page 41*), le calcul du délai $\vec{h}_{k+1}$ dépend du délai $\vec{h}_k$. Par conséquent, nous commençons à donner une valeur arbitraire au délai $\vec{h}_p$ (par exemple $\vec{0}$) et nous calculons successivement, $\vec{h}_{p-1}$ en fonction de $\vec{h}_p$, $\vec{h}_{p-2}$ en fonction de $\vec{h}_{p-1}$... et enfin $\vec{h}_1$ en fonction de $\vec{h}_2$.

Une fois ces délais calculés, nous translatons les différents domaines D'_k par $-\vec{O} = -\min_{<}(\vec{i} \in D' = \cup_{m=1}^{p}(D'_m))$ afin d'assurer que $\vec{O} = \vec{0}$. Cela revient à soustraire de tous les délais $\vec{h}_k$ le vecteur $\vec{O}$. Cette dernière étape ne modifie pas la validité de notre pavage avec fusion, car tous les calculs seront décalés par la même valeur.

Dans la suite de ce chapitre, la translation du domaine D_k par $\vec{h}_k - \vec{O}$ sera notée par $D'_k = \{\vec{i}\ :\ \vec{O}_k \le \vec{i} \le \vec{N}_k + \vec{O}_k - \vec{1}\}$.

4.2.10 Conclusion

Grâce à la modélisation du pavage comme une transformation de boucles, nous avons mis en relation les différents paramètres et nous avons donné deux solutions aux vecteurs de translations $\vec{h}_k$ afin que notre pavage avec fusion soit valide. Maintenant nous proposons deux méthodes d'allocations des tampons qui permettent d'augmenter davantage la localité des données.

4.3 Fusion avec pavage et allocation de tampon

Dans le cas de la fusion, nous avons remplacé chaque tableau A_k par un tampon circulaire et séquentiel B_k. Dans ce chapitre nous présentons deux méthodes :

1. remplacer chaque tableau A_k par un tampon circulaire et séquentiel B_k (même principe que la fusion) ;
2. remplacer chaque tableau A_k par plusieurs sous-tampons.

4.3.1 Tampon circulaire et séquentiel

Comme dans la fusion, pour maintenir les données vivantes et pour éviter de charger plusieurs fois les mêmes données, nous remplaçons les tableaux A_1, A_2, $\cdots$ et A_{p-1} par des tampons B_1, B_2, $\cdots$ et B_{p-1}. Un tampon B_k est un tableau à une dimension qui contiendra les données vivantes du tableau A_k. Chacun de ces tampons sera géré d'une manière circulaire et séquentielle (voir *chapitre 3*). Pour ranger et récupérer les éléments du tableau A_k dans le tampon B_k, nous lui associons une fonction d'accès F_k :

$$\begin{array}{rcl} \omega_2(D'_k) & \longrightarrow & N \\ F_k : \vec{l} & \longmapsto & F_k(\vec{l}) \end{array}$$

Nous notons par Taille_k la taille du tampon B_k et nous supposons que les adresses des cases de ce tampon varient de 0 à $\text{Taille}_k - 1$. Pour commencer le rangement à la première case et avoir un stockage séquentiel et circulaire, la fonction F_k doit satisfaire les deux conditions suivantes :

$$\begin{array}{rcl} F_k(\omega_2(\vec{O}_k)) & = & 0 \\ F_k(\text{succ}(\vec{l})) & = & \begin{cases} F_k(\vec{l}) + 1 \text{ si } (F_k(\vec{l}) \neq \text{Taille}_k - 1) \\ 0 \qquad\qquad \text{sinon} \end{cases} \end{array}$$

Proposition 9 *Soient $\vec{l}$ et $\vec{l'}$ deux itérations dans le domaine $\omega_2(D'_k)$ qui écrivent dans la même case du tampon B_k. Le nombre d'itérations qui sont comprises entre $\vec{l}$ et $\vec{l'}$ (ces deux itérations incluses) est supérieur à la taille du tampon B_k.*

Preuve : Même preuve que dans le cas de la fusion.

Définition 21 (Donnée vivante) *Une donnée du tableau A_{k-1} est dite vivante à une itération $\vec{l} \in \omega_2(D'_k)$ si et seulement si :*

1. *elle est produite dans le domaine $\omega_2(D'_{k-1})$ à une itération $\vec{l_1} \preceq \vec{l}$;*
2. *il existe dans le domaine $\omega_2(D'_k)$ une itération $\vec{l_2} \succeq \vec{l}$ qui la consommera.*

Définition 22 (Volume mémoire) *Le volume mémoire $M_k(\vec{l})$ correspondant à une itération $\vec{l} \in \omega_2(D'_k)$ est le nombre d'éléments du tableau A_{k-1} qui sont vivants :*

$$\begin{array}{rcl} M_k(\vec{l}) & = & |E_k(\vec{l})| \text{ avec} \\ E_k(\vec{l}) & = & \{\vec{l_1} \in \omega_2(D'_{k-1}) |\ \exists \vec{v} \in R^k \text{ et } \exists\ \vec{l_2} \in \omega_2(D'_k)\ : \\ & & \mathcal{AP}^{-1}\vec{l_1} - \vec{h}_{k-1} = \mathcal{AP}^{-1}\vec{l_2} - \vec{h}_k + \vec{v} \text{ et } \vec{l_1} \preceq \vec{l} \preceq \vec{l_2}\} \end{array}$$

Définition 23 (Borne inférieure) *La borne inférieure du domaine $\omega_2(D'_k)$ sera notée par $\vec{L}_k$ et elle est donnée par :*

$$\vec{L}_k = \omega_2(\vec{O}_k) = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{O}_k}{\vec{T}} \rfloor \\ \mathcal{R}(\vec{O}_k \bmod \vec{T}) \end{pmatrix}$$

Définition 24 (Borne supérieure) *La borne supérieure du domaine $\omega_2(D'_k)$ sera notée par $\vec{U}_k$ et est donnée par :*

$$\vec{U}_k = \omega_2(N_k + \vec{O}_k - \vec{1}) = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{N}_k + \vec{O}_k - \vec{1}}{\vec{T}} \rfloor \\ \mathcal{R}((N_k + \vec{O}_k - \vec{1}) \bmod (\vec{T})) \end{pmatrix}$$

Définition 25 (Tuile complète) *Une tuile est dite complète si le nombre d'itérations dans cette tuile est :* $T_1 * \cdots * T_i * \cdots * T_n = a_{1,1} * \cdots * a_{i,2i-1} * \cdots * a_{n,2n-1}$. *Dans le cas contraire, elle est dite incomplète.*

Pour que toutes les tuiles du domaine $\omega_2(D'_k)$ soient complètes, il est nécessaire et suffisant que :

1. $(\vec{O}_k \bmod \vec{T}) = \vec{0}$ et
2. $((N_k + \vec{O}_k - \vec{1}) \bmod (\vec{T})) = \vec{T} - \vec{1}$

Si l'une de ces conditions n'est pas satisfaite, le calcul du nombre d'itérations dans le domaine $\omega_2(D'_k)$, entre l'origine $\omega_2(\vec{O}_k)$ et une itération $\vec{l}$ quelconque, devient compliqué à cause des tuiles incomplètes. Dans ce cas, nous surapproximons le domaine $\omega_2(D'_k)$ par un domaine $Dapro_k$ tel que sa borne inférieure et sa borne supérieure seront données respectivement par :

$$\vec{L}apro_k = \begin{pmatrix} \mathcal{E}\lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor \\ \vec{0} \end{pmatrix} \text{ et } \vec{U}apro_k = \begin{pmatrix} \mathcal{E}\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor \\ \mathcal{R}\ (\vec{T}-\vec{1}) \end{pmatrix}$$

Définition 26 (Successeur) *Le successeur de l'itération $\vec{l}$ dans le domaine $Dapro_k$ sera noté par $succ(\vec{l})$ et il est défini comme suit :*

$$succ(\vec{l}) = \begin{cases} \vec{l} + (0, \cdots, 0,\ 1)^t & si\ \ l_n \neq Uapro_{k_{2n}} \\ (succ((l_1, \cdots,\ l_{2n-1})^t),\ Lapro_{k_{2n}})^t & sinon \end{cases}$$

La *définition 26* signifie que si $l_{2n} = Uapro_{k_{2n}}$, nous devons chercher le plus petit des m tel que $\forall j \in [m \cdots n]\ : l_j = Uapro_{k_j}$. Dans ce cas, le successeur de $\vec{l}$ sera donné par :

$$\begin{aligned} \text{succ}(\vec{l}) &= \vec{l} + \vec{I}nc_{k,m} \text{ avec :} \\ \vec{I}nc_{k,m} &= (0,\ 0, \cdots,\ 0,\ 1,\ -Uapro_{k_m} + Lapro_{k_m}, \cdots, -Uapro_{k_{2n}} + Lapro_{k_{2n}})^t \end{aligned}$$

Proposition 10 *Soit* $\vec{C}_k = \begin{pmatrix} \prod_{i=2}^{n}(\mathcal{E}_i\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \mathcal{E}_i\lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor + 1)\prod_{i=1}^{n}\mathcal{R}_i\ \vec{T} \\ \vdots \\ \prod_{i=n}^{n}(\mathcal{E}_i\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \mathcal{E}_i\lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor + 1)\prod_{i=1}^{n}\mathcal{R}_i\ \vec{T} \\ \prod_{i=1}^{n}\mathcal{R}_i\vec{T} \\ \vdots \\ \prod_{i=n}^{n}\mathcal{R}_i\ \vec{T} \\ 1 \end{pmatrix}$. *Quel que soit m, le produit scalaire entre $\vec{I}nc_{k,m}$ et $\vec{C}_k$ est égal à 1. C'est-à-dire que :*

$$\forall m \in [2 \cdots 2n] : \vec{I}nc_{k,m}\ \vec{C}_k = 1$$

Preuve : Nous notons par $P(m)$ cette propriété et nous montrons par récurrence que $\forall m\ :\ P(m)$ est vraie.

1. $P(2n)$ est vraie ?
   $\vec{I}nc_{k,2n}\ \vec{C}_k = (0, \cdots,\ 1, -\mathcal{R}_n(\vec{T}-\vec{1}))^t\ (\cdots, \mathcal{R}_n\ \vec{T},\ 1)^t = \mathcal{R}_n\vec{T} - \mathcal{R}_n\vec{T} + \mathcal{R}_n\vec{1} = 1$. Donc $P(2n)$ est vraie.

2. Nous supposons que $P(r)$ est vraie et nous montrons que $P(r-1)$ l'est aussi. Nous avons 3 cas possibles :
   (a) $r \geq n+2 \Rightarrow r = n+w$ $(w \geq 2)$. Dans ce cas nous avons :

$$\begin{array}{rcl} \vec{Inc}_{k,r} & = & (0, \cdots, 0, 0, 1, -\mathcal{R}_w(\vec{T}-\vec{1}), \cdots, \mathcal{R}_n(\vec{T}-\vec{1}))^t \\ \vec{Inc}_{k,r-1} & = & (0, \cdots, 0, 1, -\mathcal{R}_{w-1}(\vec{T}-\vec{1}), -\mathcal{R}_w(\vec{T}-\vec{1}), \cdots, \mathcal{R}_n(\vec{T}-\vec{1}))^t \end{array}$$

Par conséquent :

$$\begin{array}{rcl} \vec{Inc}_{k,r-1} & = & \vec{Inc}_{k,r} + (0, \cdots, 1, -\mathcal{R}_{w-1}(\vec{T}-\vec{1}) - 1, 0, \cdots, 0)^t \\ & = & \vec{Inc}_{k,r} + (0, \cdots, 1, -\mathcal{R}_{w-1}\vec{T}, 0, \cdots, 0)^t \end{array}$$

Soit $\vec{G}_{k,w-1} = (0, \cdots, 1, -\mathcal{R}_{w-1}\vec{T}, 0, \cdots, 0)^t$. Le produit scalaire $\vec{G}_{k,w-1}\ \vec{C}_k$ est égal à :

$$\prod_{i=w-1}^{n} \mathcal{R}_i\vec{T} - \mathcal{R}_{w-1}\vec{T}\prod_{i=w}^{n}\mathcal{R}_i\vec{T} = \prod_{i=w-1}^{n} \mathcal{R}_i\vec{T}_- \prod_{i=w-1}^{n} \mathcal{R}_i\vec{T} = 0$$

Le produit scalaire $\vec{Inc}_{k,r-1}\vec{C}_k = (\vec{Inc}_{k,r} + \vec{G}_{k,w-1})\ \vec{C}_k = \vec{Inc}_{k,r}\ \vec{C}_k + \vec{G}_{k,w-1}\ \vec{C}_k$. Comme $\vec{G}_{k,w-1}\ \vec{C}_k = 0$ et $\vec{Inc}_{k,r}\ \vec{C}_k = 1$ (hypothèse de récurrence), alors :

$$\vec{Inc}_{k,r-1}\vec{C}_k = 1$$

Donc $P(r-1)$ est vraie.

   (b) $r = n+1$. Dans ce cas nous avons :

$$\begin{array}{rcl} \vec{Inc}_{k,r} & = & (0, \cdots, 0, 0, 1, -\mathcal{R}_1(\vec{T}-\vec{1}), \cdots, \mathcal{R}_n(\vec{T}-\vec{1}))^t \\ \vec{Inc}_{k,r-1} & = & (0, .., 0, 1, -\mathcal{E}_n(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor), -\mathcal{R}_1(\vec{T}-\vec{1}), .., \mathcal{R}_n(\vec{T}-\vec{1}))^t \end{array}$$

Par conséquent :

$$\vec{Inc}_{k,r-1} = \vec{Inc}_{k,r} + (0, \cdots, 1, -\mathcal{E}_n(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor) - 1, 0, \cdots, 0)^t$$

Soit $\vec{G}_{k,n} = (0, \cdots, 1, -\mathcal{E}_n(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor) - 1, 0, \cdots, 0)^t$. Le produit scalaire $\vec{G}_{k,n}\ \vec{C}_k$ est égal à :

$$\prod_{i=1}^{n}(\mathcal{R}_i\vec{T})(\mathcal{E}_n(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor)+1) - \prod_{i=1}^{n}(\mathcal{R}_i\,\vec{T})(\mathcal{E}_n(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor)+1) = 0$$

Le produit scalaire $\vec{Inc}_{k,r-1}\vec{C}_k = (\vec{Inc}_{k,r} + \vec{G}_{k,n})\ \vec{C}_k = \vec{Inc}_{k,r}\ \vec{C}_k + \vec{G}_{k,n}\ \vec{C}_k$. Comme $\vec{G}_{k,n}\ \vec{C}_k = 0$ et $\vec{Inc}_{k,r}\ \vec{C}_k = 1$ (hypothèse de récurrence), alors :

$$\vec{Inc}_{k,r-1}\vec{C}_k = 1$$

Donc $P(r-1)$ est vraie.

(c) $r \leq n$. Dans ce cas nous avons :

$$\begin{aligned}\vec{Inc}_{k,r} &= (0,\cdots,\ 0,\ 0,\ 1, -\mathcal{E}_r(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor),\cdots,\mathcal{R}_n(\vec{T}-\vec{1}))^t\\ \vec{Inc}_{k,r-1} &= (0,\cdots,0,\ 1,\ -\mathcal{E}_{r-1}(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor),\cdots,\ \mathcal{R}_n(\vec{T}-\vec{1}))^t\end{aligned}$$

Par conséquent :

$$\vec{Inc}_{k,r-1} = \vec{Inc}_{k,r} + (0,\cdots,\ 1,\ -\mathcal{E}_{r-1}(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor) - 1,\ 0,\cdots,\ 0)^t$$

Soit $\vec{G}_{k,r-1} = (0,\cdots,\ 1,\ -\mathcal{E}_{r-1}(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor) - 1,\ 0,\cdots,\ 0)^t$. Le produit scalaire $\vec{G}_{k,r-1}\ \vec{C}_k$ est alors égale à :

$$\begin{aligned} &\prod_{i=r-1}^{n}(\mathcal{E}_i(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \mathcal{E}_i\ \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor)+1)\prod_{i=1}^{n}(\mathcal{R}_i\vec{T})\\ - &\prod_{i=r-1}^{n}(\mathcal{E}_i(\lfloor\frac{\vec{N}_k+\vec{O}_k-\vec{1}}{\vec{T}}\rfloor - \mathcal{E}_i\ \lfloor\frac{\vec{O}_k}{\vec{T}}\rfloor)+1)\prod_{i=1}^{n}(\mathcal{R}_i\vec{T}) = 0\end{aligned}$$

Le produit scalaire $\vec{Inc}_{k,r-1}\vec{C}_k = (\vec{Inc}_{k,r} + \vec{G}_{k,r-1})\ \vec{C}_k = \vec{Inc}_{k,r}\ \vec{C}_k + \vec{G}_{k,r-1}\ \vec{C}_k$. Comme $\vec{G}_{k,r-1}\ \vec{C}_k = 0$ et $\vec{Inc}_{k,r}\ \vec{C}_k = 1$ (hypothèse de récurrence), alors

$$\vec{Inc}_{k,r-1}\vec{C}_k = 1$$

Donc $P(r-1)$ est vraie.

□

Lemme 7 *Soit* $nbr_k(\vec{\mathrm{Lapro}}_k,\ \vec{l})$ *le nombre d'itérations dans* Dapro_k *qui sont comprises entre* $\vec{\mathrm{Lapro}}_k$ *et* $\vec{l}$. *C'est-à-dire que* $nbr_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}) = \{\vec{l'} \in \mathrm{Dapro}_k :\ \vec{\mathrm{Lapro}}_k \preceq \vec{l'} \preceq \vec{l}\}$. *Ce nombre est donné par :*

$$\vec{C}_k\ .\ (\vec{l} - \vec{\mathrm{Lapro}}_k) + 1$$

Preuve : Nous notons par $P(\vec{l})$ cette propriété et nous montrons par récurrence que $\forall\vec{l} \in \mathrm{Dapro}_k\ : P(\vec{l})$ est vraie.

1. $P(\vec{\mathrm{Lapro}}_k)$ est vraie ?
   $\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{\mathrm{Lapro}}_k) = (\vec{\mathrm{Lapro}}_k - \vec{\mathrm{Lapro}}_k)\ .\ \vec{C}_k + 1 = 1$. Donc $P(\vec{\mathrm{Lapro}}_k)$ est varie.
2. Nous supposons que $P(\vec{l})$ est vraie et nous montrons que $P(\mathrm{succ}(\vec{l}))$ l'est aussi.

   Nous savons que $\mathrm{succ}(\vec{l}) = \vec{l} + \vec{\mathrm{Inc}}_k$ avec $\vec{\mathrm{Inc}}_k = \begin{cases}(0,\cdots,\ 0,\ 1)^t & \text{si}\ \ l_n \neq \mathrm{Lapro}_{k_{2n}}\\ \vec{\mathrm{Inc}}_{k,m}\ \text{sinon}\end{cases}$

   Comme nous avons :
   (a) $\vec{\mathrm{Inc}}_{k,m}\ .\ \vec{C}_k = 1$ (*proposition 10* en *page* 47) ;

(b) $(0, \cdots, 0, 1)^t \,.\, \vec{C}_k = 1$.

Alors :

$$\vec{\mathrm{Inc}}_k \,.\, \vec{C}_k = 1$$

Comme il n'y a pas d'itérations entre $\vec{l}$ et $\mathrm{succ}(\vec{l})$ alors :

$$\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \mathrm{succ}(\vec{l})) \;=\; \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}) + 1$$

Comme $\vec{\mathrm{Inc}}_k \,.\, \vec{C}_k = 1$ alors :

$$\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \mathrm{succ}(\vec{l})) = \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}) + \vec{\mathrm{Inc}}_k \,.\vec{C}_k$$

Nous avons par hypothèse de récurrence : $\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}) = (\vec{l} - \vec{\mathrm{Lapro}}_k) \,.\, \vec{C}_k + 1$. Par conséquent :

$$\begin{aligned}\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \mathrm{succ}(\vec{l})) &= (\vec{l} - \vec{\mathrm{Lapro}}_k) \,.\, \vec{C}_k + 1 + \vec{\mathrm{Inc}}_k \,.\vec{C}_k \\ &= (\vec{l} + \vec{\mathrm{Inc}}_k - \vec{\mathrm{Lapro}}_k) \,.\, \vec{C}_k + 1 \\ &= (\mathrm{succ}(\vec{l}) - \vec{\mathrm{Lapro}}_k) \,.\, \vec{C}_k + 1\end{aligned}$$

Par conséquent, $P(\mathrm{succ}(\vec{l}))$ est vraie.

□

Lemme 8 *Soient $\vec{l}_1$ et $\vec{l}_2$ ($\vec{l}_1 \preceq \vec{l}_2$) deux itérations dans le domaine $\omega_2(D'_k)$ et $\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2)$ le nombre d'itérations dans $\omega_2(D'_k)$ qui sont comprises entre $\vec{l}_1$ et $\vec{l}_2$, c'est-à-dire que $\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2) = |\{\vec{l'} \in \omega_2(D'_k) :\ \vec{l}_1 \preceq \vec{l'} \preceq \vec{l}_2\}|$. Ce nombre est borné par :*

$$\vec{C}_k \,.\, (\vec{l}_2 - \vec{l}_1) + 1$$

Preuve : Nous savons que :

1. $\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2) = \mathrm{nbr}'_k(\vec{L}_k,\ \vec{l}_2) - \mathrm{nbr}'_k(\vec{L}_k,\ \vec{l}_1) + 1$;
2. $\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_2) = \mathrm{nbr}'_k(\vec{L}_k,\ \vec{l}_2) + a\ (a \geq 0)$;
3. $\mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_1) = \mathrm{nbr}'_k(\vec{L}_k,\ \vec{l}_1) + b\ (0 \leq b \leq a)$.

Par conséquent :

$$\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2) = \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_2) - \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_1) - (a - b) + 1$$

Comme $a \geq b$ alors

$$\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2) \leq \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_2) - \mathrm{nbr}_k(\vec{\mathrm{Lapro}}_k,\ \vec{l}_1) + 1$$

Donc :

$$\mathrm{nbr}'_k(\vec{l}_1,\ \vec{l}_2) \leq (\vec{C}_k \,.\, (\vec{l}_2 - \vec{\mathrm{Lapro}}_k) + 1) - (\vec{C}_k \,.\, (\vec{l}_1 - \vec{\mathrm{Lapro}}_k) + 1) + 1 = \vec{C}_k \,.\, (\vec{l}_2 - \vec{l}_1) + 1$$

□

Taille du tampon B_k

À une itération $\vec{l} \in \omega_2(D'_{k+1})$ dans le code de la *figure 4.4* (*page 36*), pour calculer l'élément $A_{k+1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_{k+1})$, nous utilisons m_{k+1} données du tableau A_k qui sont produites respectivement par : $\vec{l}_1$, $\vec{l}_2$,.....et $\vec{l}_{m_{k+1}}$ telles que :

$$\mathcal{AP}^{-1}\vec{l}_q - \vec{h}_k = \mathcal{AP}^{-1}\vec{l} - \vec{h}_{k+1} + \vec{v}_q^{k+1} \ (q = 1, m_{k+1}). \tag{4.4}$$

Dans le cas de la fusion, la plus ancienne des productions est toujours fixe et toutes les données produites avant cette plus ancienne production sont mortes. Par contre, dans le cas du pavage avec fusion ces deux propriétés ne sont pas vérifiées :

1. La plus ancienne des productions varie en fonction de la position de l'itération $\vec{l}$ dans la tuile (*figure 4.5*).

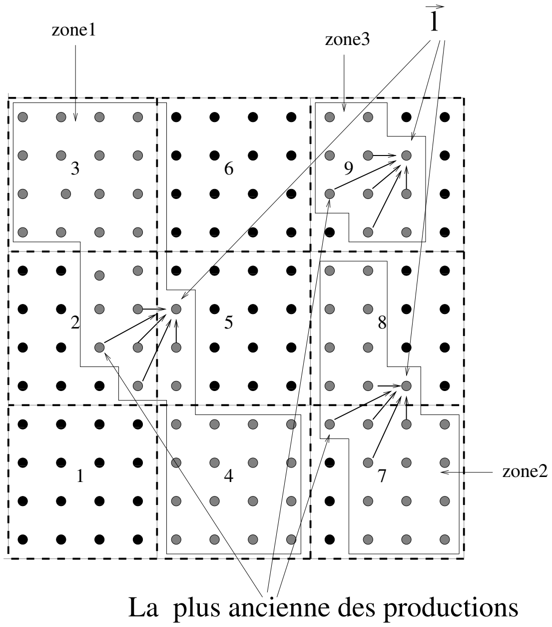


Figure 4.5: Volume mémoire en fonction de la position de l'itération $\vec{l}$ dans la tuile

2. les données qui sont produites avant la plus ancienne des productions peuvent être vivantes.

Nous cherchons à déterminer la taille du tampon B_k qui assure que, pour toute itération $\vec{l} \in \omega_2(D'_{k+1})$, toutes les données vivantes du tableau A_k doivent se trouver dans le tampon B_k, sachant que le tampon B_k est géré d'une manière circulaire et séquentielle. Pour calculer cette taille, nous devons donc parcourir tout le domaine $\omega_2(D'_{k+1})$ et chercher l'itération $\vec{l}_G$ telle que le nombre d'itérations entre cette dernière et sa plus ancienne production soit maximale.

Proposition 11 *Soit $\vec{l}_G$ une itération dans le domaine $\omega_2(D'_{k+1})$ tel que le nombre d'itérations entre $\vec{L}_G$ et sa plus ancienne production $\vec{l}_a$ soit maximale. Si la taille du tampon $B_k = \mathrm{nbr}'_k(\vec{l}_a, \vec{l}_G)$, alors quelle que soit l'itération $\vec{l} \in \omega_2(D'_{k+1})$, toutes ses données vivantes se trouvent dans le tampon B_k.*

Preuve : Nous supposons qu'il existe une itération $\vec{l} \in \omega_2(D'_{k+1})$ telle que l'une de ses données vivantes ne se trouve pas dans le tampon B_k. Par définition, cette donnée vivante est produite par $\vec{l_1} \preceq \vec{l}$ et elle sera consommée par $\vec{l_2} \succeq \vec{l}$. Comme $\vec{l_1} \preceq \vec{l}$ alors cette donnée vivante a été déjà rangée dans le tampon B_k. Donc il existe nécessairement une itération $\vec{l_3}$ ($\vec{l_1} \prec \vec{l_3} \prec \vec{l}$) qui l'a écrasée. Dans un tampon séquentiel et circulaire, si deux itérations écrivent dans la même case, alors le nombre d'itérations qui séparent ces deux itérations est supérieur à la taille de ce tampon, c'est-à-dire que le nombre d'itérations entre $\vec{l_1}$ et $\vec{l_3}$ est supérieur à $\mathrm{nbr}'_k(\vec{l_a}, \vec{l_G})$. Par conséquent, le nombre d'itérations entre $\vec{l_2}$ et l'une de ses productions ($\vec{l_1}$) est aussi supérieur à $\mathrm{nbr}'_k(\vec{l_a}, \vec{l_G})$. Contradiction, car $\mathrm{nbr}'_k(\vec{l_a},\ \vec{l_G})$ représente le plus grand nombre d'itérations que nous pouvons avoir entre une itération et sa plus ancienne production.

□

La taille du tampon B_k, qui est notée par Taille_k, va se ramener alors un problème linéaire paramétré par le vecteur $\vec{l}$ qui est modélisé comme suit :

$$\max_{\vec{l} \in \omega_2(D'_{k+1})} \left(\max_{q=1,m_{k+1}} \left(\mathrm{nbr}'_k(\vec{l_q},\ \vec{l}) \right)\right)$$
$$\left\{ \mathcal{A}\mathcal{P}^{-1}\vec{l_q} - \vec{h}_k = \mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_{k+1} + \vec{v}_q^{k+1} \right.$$

Pour calculer $\mathrm{nbr}'_k(\vec{l_q},\ \vec{l})$, nous devons en premier lieu calculer $\vec{l_q}$ en fonction de $\vec{l}$, mais ceci n'est pas possible en utilisant uniquement *l'équation 4.4* (*page 51*), car la matrice $\mathcal{A}$ n'est pas inversible. Nous savons que $\exists \vec{i} \in D'_{k+1}\ :\ \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l}$.

Par conséquent, *l'équation 4.4* (*page 51*) sera équivalente à :

$$\vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l_q} - \vec{h}_k + \vec{h}_{k+1} - \vec{v}_q^{k+1}\ (q = 1,\ m_{k+1})$$

Cela signifie que le calcul de la taille du tampon B_k va se ramener à un problème linéaire paramétré par le vecteur $\vec{i}$ qui est modélisé comme suit :

$$\max_{\vec{l} \in \omega_2(D'_{k+1})} \left(\max_{q=1,m_{k+1}} \left(nbr'_k(\vec{l_q},\ \vec{l}) \right)\right)$$
$$\begin{cases} \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l} \\ \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l_q} + \vec{h}_{k+1} - \vec{h}_k - \vec{v}_q^{k+1} \\ \vec{0} \le (l_{q,n+1}, \cdots,\ l_{q,2n})^t < \vec{T} \\ \vec{i} \in D'_{k+1} \end{cases}$$

Comme le calcul exact de $nbr'_k(\vec{l_q},\ \vec{l})$ est compliqué, nous surapproximons la taille du tampon B_k par :

$$\max_{\vec{l} \in \omega_2(D'_{k+1})} \left(\max_{q=1,m_{k+1}} \left(nbr_k(\vec{l_q},\ \vec{l}) \right)\right)$$
$$\begin{cases} \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l} \\ \vec{i} = \mathcal{A}\mathcal{P}^{-1}\vec{l_q} + \vec{h}_{k+1} - \vec{h}_k - \vec{v}_q^{k+1} \\ \vec{0} \le (l_{q,n+1}, \cdots,\ l_{q,2n})^t < \vec{T} \\ \vec{i} \in \mathrm{Dapro}_k \end{cases}$$

Fonction d'accès

Afin de ranger et de récupérer les éléments du tableau A_k dans le tampon B_k, nous lui associons une fonction d'accès F_k :

$$F_k : \begin{array}{ccc} \text{Dapro}_k & \longrightarrow & N \\ \vec{l} & \longmapsto & F_k(\vec{l}) \end{array}$$

Pour avoir un stockage séquentiel et commencer le rangement à la première position du tampon B_k, la fonction F_k doit satisfaire aux deux conditions suivantes :

$$F_k(\vec{\text{Lapro}_k}) = 0 \tag{4.5}$$

$$F_k(\text{succ}(\vec{l})) = \begin{cases} F_k(\vec{l}) + 1 \text{ si } (F_k(\vec{l}) \neq \text{Taille}_k - 1) \\ 0 \qquad\qquad \text{sinon} \end{cases} \tag{4.6}$$

Proposition 12 *Pour satisfaire aux conditions 4.5 et 4.6 il suffit de choisir :*

$$F_k(\vec{l}) = [(\vec{l} - \vec{Lapro_k}) \;.\; \vec{C_k}] \mod \; Taille_k$$

Preuve : Même preuve que dans le cas de la fusion.

□

Remarque : Les itérations appartenant à $(\text{Dapro}_k - \omega_2(D'_k))$ ne sont pas exécutées dans le code en *figure 4.4* (*page* 36). Donc, lors de la génération du code, ces itérations n'écrivent pas dans le tampon B_k. Cela revient à laisser des cases vides dans ce tampon. Cela n'a aucune influence sur la validité du code généré avec tampons donné en *figure 4.6* (*page 54*) et la séquentialité du tampon Bk, car B_k est séquentiel par rapport au domaine Dapro_k.

Génération du code

Nous considérons l'instruction S_k du code tuilé et fusionné en *figure 4.4* (*page 36*). À chaque nouvelle itération $\vec{l} \in \omega_2(D'_k)$, nous calculons l'élément $A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k)$ en fonction de m_k données du tableau A_{k-1} produites respectivement par $\vec{l_1}$, $\vec{l_2}$,.....et $\vec{l_{m_k}}$, telles que :

$$\mathcal{AP}^{-1}\vec{l_q} - \vec{h}_{k-1} = \mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_q^k \;(q = 1, m_k).$$

1. L'élément $A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k)$ sera rangé dans le tampon B_k par l'itération $\vec{l}$ à la position $F_k(\vec{l})$.
2. Les m_k éléments du tableau A_{k-1} sont déjà rangés dans le tampon B_{k-1} par les itérations $\vec{l_1}$, $\vec{l_2}$, $\cdots$ et $\vec{l_{m_k}}$, respectivement aux positions $F_{k-1}(\vec{l_1})$, $F_{k-1}(\vec{l_2})$,$\cdots$ et $F_{k-1}(\vec{l_{m_k}})$.

Par conséquent, l'instruction S_k sera remplacée par :

$$B_k(F_k(\vec{l})) = B_{k-1}(F_{k-1}(\vec{l_1})) \otimes \cdots \otimes B_{k-1}(F_{k-1}(\vec{l_{m_k}}))$$

Le code fusionné et tuilé après le remplacement des tableaux A_1,$\cdots$, A_{n-1} par des tampons B_1,$\cdots$, B_{n-1} est donné en *figure 4.6* (*page 54*).

do $\vec{l} \in \omega_2(D')$
$C_1(\vec{l})\ B_1(F_1(\vec{l})) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes .... \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$ (* s_1 *)
.
.
$C_k(\vec{l})\ B_k(F_k(\vec{l})) = B_{k-1}(F_{k-1}(\vec{i}_1)) \otimes ... \otimes B_{k-1}(F_{k-1}(\vec{l}_{m_k}))$. (* s_k *)

$C_p(\vec{l})\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = B_{p-1}(F_{p-1}(\vec{i}_1)) \otimes ... \otimes B_{p-1}(F_{p-1}(\vec{l}_{m_p}))$. (* s_p *)
enddo

Figure 4.6: Fusion avec allocation des tampons

Inconvénients de cette méthode

Cette méthode présente les inconvénients suivants :

1. À certaines itérations, le nombre de données mortes stockées dans le tampon B_k est trop élevé, car la taille du tampon B_k est égale au nombre maximal d'itérations que nous pouvons avoir entre une itération $\vec{l}$ et sa plus ancienne production et ce nombre change en fonction de la position de l'itération dans la tuile.
2. Comme nous l'avons mentionné précédemment, le calcul direct de $\vec{l_q}$ en fonction de $\vec{l}$ n'est pas possible. Par conséquent, dans le code généré nous devons programmer les fonctions de passages de $\vec{l}$ vers $\vec{i}$ et de $\vec{i}$ vers $\vec{l_q}$.

4.3.2 Allocation de plusieurs tampons

Pour pallier les problèmes liés à l'allocation des tampons circulaires et séquentiels, nous proposons de remplacer chaque tableau A_k ($1 \leq k \leq p-1$) par $2n+1$ tampons : $B_{k,1},\ B'_{k,1}, \cdots, B_{k,i},\ B'_{k,i}, \cdots, B_{k,n},\ B'_{k,n}$ et $B_{k,n+1}$. Les tampons $B_{k,i}$ et $B'_{k,i}$ sont associés à la boucle externe l_i et le tampon $B_{k,n+1}$ est associé aux n boucles internes $(l_{n+1}, \cdots, l_{2n})$.

Soit $\vec{l}$ une itération dans le domaine $\omega_2(D'_{k+1})$. Les données vivantes du tableau A_k à l'itération $\vec{l}$ sont définies comme suit :

$$\begin{aligned} E_{k+1}(\vec{l}) \ &= \ \{\vec{l'} \in \omega_2(D'_k) |\ \exists \vec{v} \in R^{k+1} \text{ et } \exists\ \vec{l}'' \in \omega_2(D'_{k+1}) \ : \\ & \mathcal{A}\mathcal{P}^{-1}\vec{l'} - \vec{h}_k = \mathcal{A}\mathcal{P}^{-1}\vec{l}'' - \vec{h}_{k+1} + \vec{v} \text{ et } \vec{l'} \preceq \vec{l} \preceq \vec{l}''\} \end{aligned}$$

C'est-à-dire que :

$$\begin{aligned} E_{k+1}(\vec{l}) \ &= \ \{\vec{l'} \in \omega_2(D'_k) |\ \exists\ \vec{l}'' \in \omega_2(D'_{k+1}) \ : \\ & \vec{l'} \preceq \vec{l} \preceq \vec{l}'' \text{ et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'}\} \end{aligned}$$

Notre objectif est de partitionner cet ensemble en plusieurs sous-ensembles. Chacun de ces derniers va correspondre à un tampon.

4.3.3 Profondeur des nids du code en entrée est 2

Vu la difficulté du problème, nous traitons d'abord le cas des séquences de nids de boucles de profondeur 2 et nous considérons un ordonnancement particulier (la matrice $\mathcal{P}$ est fixe) :

1. les différents nids du code en entrée donné en *figure 4.1* (*page 32*) sont de profondeur $n = 2$;
2. la matrice de permutation $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Comme la profondeur des différents nids est $n = 2$, alors la matrice du pavage $\mathcal{A}$ sera de taille $(n,\ 2n) = (2,\ 4)$ et elle aura la forme suivante :

$$\mathcal{A} = \begin{pmatrix} a_{1,1} & 1 & 0 & 0 \\ 0 & 0 & a_{2,3} & 1 \end{pmatrix}$$

Avec :

1. $a_{1,1} = T_1$ est la taille des tuiles sur l'axe i ;
2. $a_{2,3} = T_2$ est la taille des tuiles sur l'axe j.

Après l'application du pavage défini par les matrices $\mathcal{A}$ et $\mathcal{P}$, nous obtenons le code en *figure 4.7* :

1. l_1 permet de parcourir les tuiles sur l'axe i ;
2. l_2 permet de parcourir les tuiles sur l'axe j ;
3. l_3 permet de parcourir les itérations à l'intérieur de tuiles sur l'axe i ;
4. l_4 permet de parcourir les itérations à l'intérieur de tuiles sur l'axe j.

do(l_1)
 do(l_2)
  do(l_3)
   do(l_4)

$$C_1(\vec{l}) : A_1(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}^1_1) \otimes .. \otimes A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}^1_{m_1})$$

.

.

$$C_k(\vec{l}) : A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}^k_1) \otimes .. \otimes A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}^k_{m_k})$$

.

.

$$C_p(\vec{l}) : A_p(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}^p_1) \otimes .. \otimes A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}^p_{m_p})$$

   enddo
  enddo
 enddo
enddo

Figure 4.7: Application du pavage défini par $\mathcal{A}$ et $\mathcal{P}$ sur un code dont la profondeur des nids est $n = 2$

Position d'une itération production :

À chaque itération $\vec{l} \in \omega_2(D'_{k+1})$ du code en *figure 4.7*, nous calculons un nouvel élément du tableau A_{k+1} en fonction de m_{k+1} éléments du tableau A_k qui sont produits respectivement par des itérations $\vec{l}^1, \cdots, \vec{l}^{m_{k+1}}$. Soit $\vec{d}^{k,q}$ la dépendance entre $\vec{i}^q = \mathcal{AP}^{-1}(\vec{l}^q)$ et $\vec{i} = \mathcal{AP}^{-1}(\vec{l})$; la dépendance par rapport aux axes i et j. Dans la suite nous supposons que tous les $d_m^{k,q}$ sont compris entre 0 et $T_m = a_{m,2m-1}$, c'est-à-dire que la projection de n'importe quelle dépendance sur n'importe quel axe est positive (les boucles sont complétement permutables) et ne dépasse pas la taille des tuiles sur cet axe.

Proposition 13 *Soit $\vec{l'} \in \omega_2(D'_k)$ une itération qui produit un élément du tableau A_k qui sera consommé par une itération $\vec{l} \in \omega_2(D'_{k+1})$. Comme $\forall m,\ \forall q\ : 0 \leq d_m^{k,q} \leq T_m = a_{m,2m-1}$, alors nous avons les contraintes :*

1. $l'_1 \geq l_1 - 1$;
2. $l'_2 \geq l_2 - 1$.

Preuve : Soient les itérations $\vec{i'}$ et $\vec{i}$ dans l'espace initial qui correspondent aux itérations $\vec{l}$ et $\vec{l'}$. C'est-à-dire que $\vec{i} = \mathcal{AP}^{-1}\vec{l}$ et $\vec{i'} = \mathcal{AP}^{-1}\vec{l'}$. Nous savons que :

1. $\vec{i'} = \vec{i} - \vec{d}$ avec $\vec{d} \geq \vec{O}$;
2. $l_1 = \lfloor \frac{i_1}{a_{1,1}} \rfloor$;
3. $l_2 = \lfloor \frac{i_2}{a_{2,3}} \rfloor$;
4. $l'_1 = \lfloor \frac{i'_1}{a_{1,1}} \rfloor = \lfloor \frac{i_1 - d_1}{a_{1,1}} \rfloor$;
5. $l'_2 = \lfloor \frac{i'_2}{a_{2,3}} \rfloor = \lfloor \frac{i_2 - d_2}{a_{2,3}} \rfloor$.

Comme nous avons supposé que $0 \leq d_1 \leq a_{1,1}$ et $0 \leq d_2 \leq a_{2,3}$ alors :

1. $\lfloor \frac{i_1 - d_1}{a_{1,1}} \rfloor \geq \lfloor \frac{i_1}{a_{1,1}} \rfloor - 1 \Rightarrow l'_1 \geq l_1 - 1$;
2. $\lfloor \frac{i_2 - d_2}{a_{2,3}} \rfloor \geq \lfloor \frac{i_2}{a_{2,3}} \rfloor - 1 \Rightarrow l'_2 \geq l_2 - 1$.

□

Proposition 14 *Soit $\vec{l'}$ une itération qui produit un élément du tableau A_k qui sera consommé par une itération $\vec{l}$. L'itération $\vec{l'}$ peut se trouver dans trois ensembles possibles :*

$$
\begin{aligned}
Z_{k,1}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1 - 1\}\ ; \\
Z_{k,2}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1 \text{ et } l'_2 = l_2 - 1\}\ ; \\
Z_{k,3}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1,\ l'_2 = l_2 \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t\}.
\end{aligned}
$$

Preuve : Comme $\vec{l'}$ produit un élément qui est consommé par $\vec{l}$ alors $\vec{l'} \preceq \vec{l}$. D'après la définition de l'opérateur $\preceq$, nous avons les trois cas suivants :

1. $l'_1 < l_1$. D'après *la proposition 13* (*page 56*) : $l'_1 = l_1 - 1$. C'est-à-dire que :
$$\vec{l'} \in Z_{k,1}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1 - 1\}$$
2. $l'_1 = l_1$ et $l'_2 < l_2$. D'après *la proposition 13* (*page 56*) : $l'_2 = l_2 - 1$. C'est-à-dire que :
$$\vec{l'} \in Z_{k,2}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1 \text{ et } l'_2 = l_2 - 1\}$$
3. $l'_1 = l_1$, $l'_2 = l_2$ et $(l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t$. C'est-à-dire que :
$$\vec{l'} \in Z_{k,3}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1,\ l'_2 = l_2 \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t\}$$

□

Partition des données vivantes :

Afin d'allouer plusieurs tampons, nous décomposons l'ensemble des données vivantes de chaque tableau en plusieurs sous-ensembles. Chacun de ces sous-ensembles va correspondre à un tampon.

Proposition 15 *Soit $\vec{l}$ une itération dans le domaine $\omega_2(D'_{k+1})$ et $E_{k+1,1}(\vec{l})$, $E_{k+1,2}(\vec{l})$ et $E_{k+1,3}(\vec{l})$ des ensembles qui sont définis comme suit :*

1. $E_{k+1,1}(\vec{l}) = \{\vec{l'} \in \omega_2(D'_k) | \exists \, \vec{l}\,'' \in \omega_2(D'_{k+1}) \; : l'_1 = l_1 - 1, \; l_1'' = l_1$ *et* $\vec{l}\,''$ *consomme la donnée produite par* $\vec{l'} \}$;
2. $E_{k+1,2}(\vec{l}) = \{\vec{l'} \in \omega_2(D'_k) | \exists \, \vec{l}\,'' \in \omega_2(D'_{k+1}) \; : l'_1 = l_1, \; l'_2 = l_2 - 1, \; l_1'' = l_1, \; l_2'' = l_2$ *et* $\vec{l}\,''$ *consomme la donnée produite par* $\vec{l'} \}$;
3. $E_{k+1,3}(\vec{l}) = \{\vec{l'} \in \omega_2(D'_k) | \exists \, \vec{l}\,'' \in \omega_2(D'_{k+1}) \; : l'_1 = l_1, l'_2 = l_2, (l'_3, \; l'_4)^t \preceq (l_3, \; l_4)^t, \; l_1'' = l_1, \; l_2'' = l_2, (l_3'', \; l_4'')^t \succeq (l_3, \; l_4)^t,$ *et* $\vec{l}\,''$ *consomme la donnée produite par* $\vec{l'} \}$.

Les cardinaux de $E_{k+1,1}(\vec{l})$, $E_{k+1,2}(\vec{l})$ et $E_{k+1,3}(\vec{l})$ sont respectivement donnés par $N_2^k * d_1^{k,max}$, $T_1 * d_2^{k,max}$ et $T_2 * d_1^{k,max}$. Avec :

1. *$T_1 = a_{1,1}$ est la taille des tuiles sur l'axe i ;*
2. *$T_2 = a_{2,3}$ est la taille des tuiles sur l'axe j ;*
3. *$d_1^{k,max}$ est la plus grande projection sur l'axe i de toutes les dépendances de l'instruction S_k vers l'instruction S_{k+1} du code en* figure 4.7 *(*page 55*) ;*
4. *$d_2^{k,max}$ est la plus grande projection sur l'axe j de toutes les dépendances de l'instruction S_k vers l'instruction S_{k+1} du code en* figure 4.7 *(*page 55*) ;*
5. *N_2^k est le nombre d'itérations de la deuxième boucle du nid k du code en* figure 4.1 *(*page 32*).*

Preuve : Soit $\vec{d}$ la dépendance entre $\vec{i'} = \mathcal{AP}^{-1}\vec{l'}$ et $\vec{i}\,'' = \mathcal{AP}^{-1}\vec{l}\,''$ ($\vec{i'} + \vec{d} = \vec{i}\,''$).

1. $|E_{k+1,1}(\vec{l})|$?

   Nous savons que :

$$\begin{array}{rcl} i'_1 & = & a_{1,1} * l'_1 + l'_3 \\ i'_2 & = & a_{2,3} * l'_2 + l'_4 \\ i_1'' & = & a_{1,1} * l_1'' + l_3'' \\ i_2'' & = & a_{2,3} * l_2'' + l_4'' \end{array}$$

   Comme $\vec{i'} + \vec{d} = \vec{i}\,''$ alors :

$$\begin{array}{rcl} a_{1,1} * l'_1 + l'_3 + d_1 & = & a_{1,1} * l_1'' + l_3'' \\ a_{2,3} * l'_2 + l'_4 + d_2 & = & a_{2,3} * l_2'' + l_4'' \end{array}$$

   D'après la définition de l'ensemble $E_{k+1,1}(\vec{l})$: $l'_1 = l_1'' - 1$.
   Comme $l'_1 = l_1'' - 1$ et $a_{1,1} * l'_1 + l'_3 + d_1 = a_{1,1} * l_1'' + l_3''$ alors :

$$l'_3 = l_3'' + a_{1,1} - d_1$$

Nous savons que $0 \leq l_3^{''} \leq a_{1,1} - 1$. Par conséquent :

$$a_{1,1} - d_1 \leq l_3' \leq 2 * a_{1,1} - d_1 - 1$$

Comme $l_3' \leq a_{1,1} - 1$ et $0 \leq d_1 \leq a_{1,1}$ alors :

$$a_{1,1} - d_1 \leq l_3' \leq a_{1,1} - 1$$

Comme nous cherchons à prendre toutes les itérations $\vec{l'}$ qui sont susceptibles d'appartenir à $E_{k+1,1}(\vec{l})$ alors la contrainte précédente peut être remplacée par :

$$a_{1,1} - d_1^{k,max} \leq l_3' \leq a_{1,1} - 1$$

Par conséquent, les seules conditions que doit satisfaire $\vec{l'}$ sont :

(a) $l_1' = l_1 - 1$: C'est-à-dire que toutes les tuiles qui ont la coordonnée $l_1 - 1$ sur l'axe i sont concernées. Le nombre total de ces tuiles est $N_2^k / a_{2,3}$.

(b) $a_{1,1} - d_1^{k,max} \leq l_3' \leq a_{1,1} - 1$: C'est-à-dire que dans la même tuile, les itérations qui sont concernées sont celles qui satisfont $a_{1,1} - d_1^{k,max} \leq l_3' \leq a_{1,1} - 1$. Le nombre total de ces itérations est $a_{2,3} * d_1^{k,max}$.

Par conséquent, le nombre total d'itérations $\vec{l'} \in E_{k+1,1}(\vec{l})$ est borné par :

$$(N_2^k / a_{2,3}) * (a_{2,3} * d_1^{k,max}) = N_2^k * d_1^{k,max}$$

2. $|E_{k+1,2}(\vec{l})|$?

De la même manière que le cas précédent, nous obtenons :

$$\begin{aligned} a_{1,1} * l_1' + l_3' + d_1 &= a_{1,1} * l_1^{''} + l_3^{''} \\ a_{2,3} * l_2' + l_4' + d_2 &= a_{2,3} * l_2^{''} + l_4^{''} \end{aligned}$$

D'après la définition de l'ensemble $E_{k+1,2}(\vec{l})$:

$$l_1' = l_1^{''} \text{ et } l_2' = l_2^{''} - 1$$

Par conséquent :

$$\begin{aligned} l_3' &= l_3^{''} - d_1 \\ l_4' &= l_4^{''} + a_{2,3} - d_2 \end{aligned}$$

Nous savons que $0 \leq l_3^{''} \leq a_{1,1} - 1$ et $0 \leq l_4^{''} \leq a_{2,3} - 1$, donc :

$$\begin{aligned} -d_1 &\leq l_3' \leq a_{1,1} - d_1 - 1 \\ a_{2,3} - d_2 &\leq l_4' \leq 2 * a_{2,3} - d_2 - 1 \end{aligned}$$

Comme $l_3' \geq 0$, $0 \leq d_2 \leq a_{2,3}$, $0 \leq d_1 \leq a_{1,1}$ et $l_4^{''} \leq a_{2,3}$ alors :

$$\begin{aligned} 0 &\leq l_3' \leq a_{1,1} - 1 \\ a_{2,3} - d_2 &\leq l_4' \leq a_{2,3} - 1 \end{aligned}$$

Comme nous cherchons à prendre toutes les itérations qui sont susceptibles d'appartenir à $E_{k+1,2}(\vec{l})$, les deux contraintes précédentes peuvent être remplacées par :

$$\begin{aligned} 0 &\leq l_3' \leq a_{1,1} - 1 \\ a_{2,3} - d_2^{k,max} &\leq l_4' \leq a_{2,3} - 1 \end{aligned}$$

Par conséquent, les seules conditions que doit satisfaire $\vec{l'}$ sont :

(a) $l'_1 = l_1$ et $l'_2 = l_2 - 1$. C'est-à-dire que les itérations $\vec{l'}$ qui sont concernées sont situées dans la même tuile.

(b) $0 \leq l'_3 \leq a_{1,1} - 1$ et $a_{2,3} - d_2^{k,max} \leq l'_4 \leq a_{2,3} - 1$. C'est-à-dire que dans cette même tuile, les itérations $\vec{l'}$ qui sont concernées sont celles qui satisfont les deux contraintes précédentes. Le nombre total des couples d'entiers satisfaisant ces deux contraintes est donné par $a_{1,1} * d_2^{k,max}$

Par conséquent, le nombre total d'itérations $\vec{l'} \in E_{k+1,2}(\vec{l})$ est borné par :

$$a_{1,1} * d_2^{k,max} = T_1 * d_2^{k,max}$$

3. $|E_{k+1,3}(\vec{l})|$?

La définition de $|E_{k+1,3}(\vec{l})|$ revient à calculer le nombre de données vivantes dans la même tuile. Soit $(l_3,\ l_4)^t$ les cordonnées d'une itération quelconque dans une tuile donnée. La plus ancienne itération dans cette même tuile qui peut produire une donnée qui sera utilisée par l'itération $(l_3,\ l_4)^t$ est $(l'_3,\ l'_4)^t$ avec $l'_3 = l_3 - d_1^{k,max}$. De la même manière que la fusion, nous pouvons vérifier facilement que toutes les données vivantes sont produites entre $(l'_3,\ l'_4)^t$ et $(l_3,\ l_4)^t$. Le nombre d'itérations (données qui sont produites) entre ces deux itérations est borné par $a_{2,3} * (d_1^{k,max} + 1) = T_2 * (d_1^{k,max} + 1)$. Par conséquent le nombre d'itérations $\vec{l'} \in E_{k+1,3}(\vec{l})$ est borné par :

$$a_{2,3} * (d_1^{k,max} + 1) = T_2 * (d_1^{k,max} + 1)$$

□

À chaque itération, nous avons choisi d'utiliser deux tampons différents pour lire les données vivantes et pour écrire les nouvelles données vivantes. N'utiliser qu'un tampon pour ces deux ensembles en aurait considérablement compliqué la fonction d'accès. Au niveau d'une tuile, nous utilisons qu'un seul tampon.

Théorème 7 *Pour toute itération $\vec{l}$ dans le domaine $\omega_2(D'_{k+1})$, le tableau A_k peut être remplacé par 5 tampons :*

1. *un tampon $B_{k,1}$ de taille $N_2^k * d_1^{k,max}$ qui contiendra les données produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1 - 1$ et $l_1'' = l_1$;*
2. *un tampon $B'_{k,1}$ de taille $N_2^k * d_1^{k,max}$ qui contiendra les données produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$ et $l_1'' = l_1 + 1$;*
3. *un tampon $B_{k,2}$ de taille $T_1 * d_2^{k,max}$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$, $l'_2 = l_2 - 1$, $l_1'' = l_1$ et $l_2'' = l_2$;*
4. *un tampon $B'_{k,2}$ de taille $T_1 * d_2^{k,max}$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$, $l'_2 = l_2$, $l_1'' = l_1$ et $l_2'' = l_2 + 1$;*
5. *un tampon $B_{k,3}$ de taille $T_2 * d_1^{k,max}$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$, $l'_2 = l_2$, $l_1'' = l_1$ et $l_2'' = l_2$.*

Preuve : Nous devons montrer que n'importe quelle donnée vivante du tableau A_k à n'importe la quelle itération $\vec{l}$ doit se trouver dans les tampons $B_{k,1}$, $B'_{k,1}$, $B_{k,2}$, $B'_{k,2}$ ou $B_{k,3}$.

Soit $E_{k+1}(\vec{l})$ l'ensemble des données vivantes du tableau A_k à une itération $\vec{l}$:

$$\begin{aligned} E_{k+1}(\vec{l}) \ = \ \{\vec{l'} \in \omega_2(D'_k) | \ \exists \ \vec{l}'' \in \omega_2(D'_{k+1}) \ : \\ \vec{l'} \preceq \vec{l} \preceq \vec{l}'' \text{ et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'} \ \} \end{aligned}$$

D'après la définition de l'opérateur $\preceq$, la relation $\vec{l'} \preceq \vec{l}$ donne les cas suivants :

1. $l'_1 < l_1$. Comme $\vec{l'} \preceq \vec{l} \preceq \vec{l}''$ alors :

$$l'_1 \leq l_1 \leq l_1'' \Rightarrow l'_1 < l_1 \leq l_1''$$

Comme $\vec{l}''$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 13* (*page 56*) :

$$l'_1 \geq l_1'' - 1$$

Comme $l'_1 \geq l_1'' - 1$ et $l'_1 < l_1 \leq l_1''$ alors :

$$l_1'' - 1 \leq l'_1 < l_1 \leq l_1''$$

Cela signifie que :

$$l'_1 = l_1 - 1 \text{ et } l_1 = l_1''$$

C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E_{k+1,1}(\vec{l}) \ = \ \{\vec{l'} \in \omega_2(D'_k) | \ \exists \ \vec{l}'' \in \omega_2(D'_{k+1}) \ : l'_1 = l_1 - 1, \ l_1'' = l_1 \text{ et} \\ \vec{l}'' \text{ consomme la donnée produite par } \vec{l'} \ \} \end{aligned}$$

D'après la *proposition 15* (*page 57*), l'ensemble $E_{k+1,1}(\vec{l})$ est borné par $N_2^k * d_1^{k,max}$. Par conséquent, toutes les donnés du tableau A_k qui sont produites par des $\vec{l'} \in E_{k+1,1}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,1}$ de taille $N_2^k * d_1^{k,max}$ (chaque $\vec{l'}$ produit une seule donnée).

2. $l'_1 = l_1$ et $l'_2 < l_2$. Comme $\vec{l'} \preceq \vec{l}''$ alors $l'_1 \leq l_1''$.
Comme $\vec{l}''$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 13* (*page 56*) :

$$l'_1 \geq l_1'' - 1$$

Par conséquent :

$$l'_1 \leq l_1'' \leq l'_1 + 1$$

Donc nous avons deux sous-cas :

(a) $l_1'' = l'_1$.
Comme $l_1'' = l'_1 = l_1$ et $\vec{l} \preceq \vec{l}''$ alors :

$$l_2'' \geq l_2$$

Comme $\vec{l}^{''}$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 13* (*page 56*) :

$$l_2^{''} \leq l_2' + 1$$

Comme $l_2' < l_2$, $l_2^{''} \geq l_2$ et $l_2^{''} \leq l_2' + 1$ alors

$$l_2' < l_2 \leq l_2^{''} \leq l_2' + 1$$

Cela signifie que

$$l_2' = l_2 - 1 \text{ et } l_2^{''} = l_2$$

C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E_{k+1,2}(\vec{l}) \;&=\; \{\vec{l'} \in \omega_2(D_k') |\; \exists\, \vec{l}^{''} \in \omega_2(D_{k+1}') \;: l_1' = l_1,\; l_2' = l_2 - 1,\; l_1^{''} = l_1, \\ &\quad l_2^{''} = l_2 \;\text{ et } \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'}\;\} \end{aligned}$$

D'après la *proposition 15* (*page 57*), l'ensemble $E_{k+1,2}(\vec{l})$ est borné par $T_1 * d_2^{k,max}$. Par conséquent, toutes les données du tableau A_k produites par des $\vec{l'} \in E_{k+1,2}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,2}$ de taille $T_1 * d_2^{k,max}$.

(b) $l_1^{''} = l_1' + 1$ C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E_{k+1,1}'(\vec{l}) \;&=\; \{\vec{l'} \in \omega_2(D_k') |\; \exists\, \vec{l}^{''} \in \omega_2(D_{k+1}') \;: l_1' = l_1,\; l_1^{''} = l_1 + 1 \text{ et} \\ &\quad \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'}\;\} \end{aligned}$$

En posant $\vec{m} = \vec{l} + (1,\ 0,\ 0,\ 0)^t$, nous obtenons $E_{k+1,1}'(\vec{l}) = E_{k+1,1}(\vec{m})$ qui est borné par $N_2^k * d_1^{k,max}$. Par conséquent, toutes les donnés du tableau A_k produites par des $\vec{l'} \in E_{k+1,1}'$ peuvent être stockées dans un tampon $B_{k,1}'$ de taille $N_2^k * d_1^{k,max}$.

3. $l_1' = l_1$ et $l_2' = l_2$. Comme $\vec{l'} \preceq \vec{l}^{''}$ alors $l_1' \leq l_1^{''}$.
Comme $\vec{l}^{''}$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 13* (*page 56*) :

$$l_1' \geq l_1^{''} - 1$$

Par conséquent :

$$l_1' \leq l_1^{''} \leq l_1' + 1$$

Donc nous avons deux sous-cas :

(a) $l_1^{''} = l_1'$.
Comme $l_1 = l_1^{''}$ et $\vec{l} \preceq \vec{l}^{''}$ alors $l_2^{''} \geq l_2$. Donc nous avons deux cas :

i. $l_2^{''} = l_2$. Comme $\vec{l'} \preceq \vec{l} \preceq \vec{l}^{''}$, $l_1' = l_1$, $l_2' = l_2$, $l_1^{''} = l_1$ et $l_2^{''} = l_2$, alors :

$$(l_3',\ l_4')^t \preceq (l_3,\ l_4)^t \preceq (l_3^{''},\ l_4^{''})^t$$

C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E_{k+1,3}(\vec{l}) \;&=\; \{\vec{l'} \in \omega_2(D_k') |\; \exists\, \vec{l}^{''} \in \omega_2(D_{k+1}') \;: l_1' = l_1,\; l_2' = l_2,\; l_1^{''} = l_1, \\ &\quad l_2^{''} = l_2,\; (l_3',\ l_4')^t \preceq (l_3,\ l_4)^t \preceq (l_3^{''},\ l_4^{''})^t \text{ et } \vec{l}^{''} \text{ consomme} \\ &\quad \text{la donnée produite par } \vec{l'}\;\} \end{aligned}$$

D'après la *proposition 15* (*page 57*), l'ensemble $E_{k+1,3}(\vec{l})$ est borné par $T_2 * d_1^{k,max}$. Par conséquent, toutes les données du tableau A_k produites par des $\vec{l'} \in E_{k+1,3}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,3}$ de taille $T_2 * d_1^{k,max}$.

ii. $l_2^{''} > l_2$.
Comme $\vec{l}^{''}$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 13* (*page 56*) :

$$l_2^{''} \leq l'_2 + 1$$

Par conséquent :

$$l_2 < l_2^{''} \leq l'_2 + 1 = l_2 + 1$$

Cela signifie que :

$$l_2^{''} = l_2 + 1 \text{ et } l'_2 = l_2$$

C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E'_{k+1,2}(\vec{l}) \quad = \quad & \{\vec{l'} \ \in \omega_2(D'_k) | \ \exists \ \vec{l}^{''} \in \omega_2(D'_{k+1}) \ : l'_1 = l_1, \ l'_2 = l_2 \ , l_1^{''} = l_1, \\ & l_2^{''} = l_2 + 1 \ \text{ et } \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \ \} \end{aligned}$$

En posant $\vec{m} = \vec{l} + (0,\ 1,\ 0,\ 0)^t$, nous obtenons $E'_{k+1,2}(\vec{l}) = E_{k+1,2}(\vec{m})$ qui est borné par $T_1 * d_2^{k,max}$. Par conséquent, toutes les donnés du tableau A_k produites par des $\vec{l'} \in E'_{k+1,2}(\vec{l})$ peuvent être stockées dans un tampon $B'_{k,2}$ de taille $T_1 * d_2^{k,max}$.

(b) $l_1^{''} = l'_1 + 1$. C'est-à-dire que :

$$\begin{aligned} \vec{l'} \in E'_{k+1,1}(\vec{l}) \quad = \quad & \{\vec{l'} \ \in \omega_2(D'_k) | \ \exists \ \vec{l}^{''} \in \omega_2(D'_{k+1}) \ : l'_1 = l_1, \ l_1^{''} = l_1 + 1 \text{ et} \\ & \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \ \} \end{aligned}$$

En posant $\vec{m} = \vec{l} + (1,\ 0,\ 0,\ 0)^t$, nous obtenons $E'_{k,1}(\vec{l}) = E_{k,1}(\vec{m})$ qui est borné par $N_2^k * d_1^{k,max}$. Par conséquent, toutes les données du tableau A_k produites par des $\vec{l'} \in E'_{k+1,1}$ peuvent être stockées dans un tampon $B'_{k,1}$ de taille $N_2^k * d_1^{k,max}$.

□

4.3.4 Génération du code en dimension 2

Soit $\vec{l}$ une itération dans le domaine $\omega_2(D'_{k+1})$. D'après la *proposition 14* (*page 56*), une itération $\vec{l'} \in \omega_2(D'_k)$ qui produit une donnée du tableau A_k qui sera consommée par l'itération $\vec{l}$ peut se trouver dans 3 zones possibles :

1. $Z_{k,1}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1 - 1\}$
2. $Z_{k,2}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1 \ et \ l'_2 = l_2 - 1\}$
3. $Z_{k,3}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1, \ l'_2 = l_2 \ \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t\}$

Et d'après le *théorème 7* (*page 59*), les données qui sont produites dans :

1. $Z_{k,1}(\vec{l})$ sont stockées dans le tampon $B_{k,1}$;
2. $Z_{k,2}(\vec{l})$ sont stockées dans le tampon $B_{k,2}$;

3. $Z_{k,3}(\vec{l})$ sont stockées dans le tampon $B_{k,3}$.

Mais chaque nouvelle donnée du tableau A_k produite par une itération $\vec{l'}$ sera sauvegardée dans l'un (ou plusieurs) des tampons suivants : $B'_{k,1}$, $B'_{k,2}$ ou $B_{k,3}$ (*théorème 7* en *page 59*).

Donc, à une itération $\vec{l}$, pour utiliser une donnée du tableau A_k produite par une itération $\vec{l'}$, nous devons la récupérer dans les tampons $B_{k,1}$, $B_{k,2}$ ou $B_{k,3}$. Mais, comme l'itération $\vec{l'}$ sauvegarde cette donnée dans les tampons $B'_{k,1}$, $B'_{k,2}$ ou $B_{k,3}$, nous devons :

1. transférer tous les éléments du tampon $B'_{k,2}$ dans le tampon $B_{k,2}$ lorsque nous passons d'une tuile vers la prochaine série. Car la donnée sauvegardée dans le tampon $B'_{k,2}$ à une itération $\vec{l'}$ sera récupérée par une itération $\vec{l}$ ($l_1 = l'_1$ et $l_2 = l'_2+1$) dans le tampon $B_{k,2}$;
2. transférer tous les éléments du tampon $B'_{k,1}$ dans le tampon $B_{k,1}$ lorsque nous passons d'une colonne de tuiles vers la prochaine. Car la donnée sauvegardée dans le tampon $B'_{k,1}$ à une itération $\vec{l'}$, sera récupérée par une itération $\vec{l}$ ($l_1 = l'_1 + 1$) dans le tampon $B_{k,1}$.

Cette méthode de génération du code n'est pas efficace, à cause des copies des données de $B'_{k,1}$ dans $B_{k,1}$ et de $B'_{k,2}$ dans $B_{k,2}$. Pour pallier ce problème, il suffit de fusionner les tampons, pour les utiliser en *flip-flop* :

1. $B_{k,1}$ et $B'_{k,1}$ dans un seul tampon qui sera noté par $BT_{k,1}[2]$ tel que $|BT_{k,1}[0]| = |BT_{k,1}[1]| = |B_{k,1}| = |B'_{k,1}|$. Dans ce cas, une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,1}$ va la stocker dans $BT_{k,1}[l_1 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,1}$, elle va la récupérer dans $BT_{k,1}[(l_1 - 1) \bmod 2]$.
2. $B_{k,2}$ et $B'_{k,2}$ en un seul tampon qui sera noté par $BT_{k,2}[2]$ tel que $|BT_{k,2}[0]| = |BT_{k,2}[1]| = |B_{k,2}| = |B'_{k,2}|$. Dans ce cas une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,2}$ va la stocker dans $BT_{k,2}[l_2 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,2}$, elle va la récupérer dans $BT_{k,2}[(l_2 - 1) \bmod 2]$.

Il y a plusieurs méthodes pour gérer ces différents tampons que nous n'avons pas mentionnées dans ce chapitre ; il sera intéressant dans des travaux futurs de tester et de comparer expérimentalement ces différentes méthodes.

4.3.5 Cas Général (profondeur des nids quelconque)

Dans le cas général, c'est-à-dire que lorsque la profondeur des nids et la matrice de permutation $\mathcal{P}$ sont quelconques, le tableau A_k sera remplacé par $2n+1$ tampons. Le *théorème 8* définit ces différents tampons et leurs tailles.

Théorème 8 *Pour tout $\vec{l}$ dans le domaine $\omega_2(D'_{k+1})$, le tableau A_k peut être remplacé par $2n+1$ tampons :*

- *un tampon $B_{k,1}$ de taille $\mathcal{E}_1\ \vec{d}^{k,max} * \prod_{i=2}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1 - 1$ et $l_1'' = l_1$;*
- *un tampon $B'_{k,1}$ de taille $\mathcal{E}_1\ \vec{d}^{k,max} * \prod_{i=2}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$ et $l_1'' = l_1 + 1$;*

$\vdots$

$\vdots$

– *un tampon* $B_{k,m}$ *de taille* $\prod_{i=1}^{m-1}(\mathcal{E}_i\ \vec{T}) * (\mathcal{E}_m\ \vec{d}^{k,max}) * \prod_{i=m+1}^{n}(\mathcal{E}_i\ \vec{N}^k)$ *qui contiendra les données du tableau* A_k *produites par des* $\vec{l'}$ *et qui seront consommées par des* $\vec{l}''$ *telles que* $l'_1 = l_1, \cdots, l'_{m-1} = l_{m-1}$, $l'_m = l_m - 1$, $l''_1 = l_1, \cdots$ *et* $l''_m = l_m$.
– *un tampon* $B'_{k,m}$ *de taille* $\prod_{i=1}^{m-1}(\mathcal{E}_i\ \vec{T}) * (\mathcal{E}_m\ \vec{d}^{k,max}) * \prod_{i=m+1}^{n}(\mathcal{E}_i\ \vec{N}^k)$ *qui contiendra les données du tableau* A_k *produites par des* $\vec{l'}$ *et qui seront consommées par des* $\vec{l}''$ *telles que* $l'_1 = l_1, \cdots, l'_m = l_m$, $l''_1 = l_1, \cdots, l''_{m-1} = l_{m-1}$ *et* $l''_m = l_m + 1$;

⋮

⋮

– *un tampon* $B_{k,n}$ *de taille* $\prod_{i=1}^{n-1}(\mathcal{E}_i\ \vec{T}) * (\mathcal{E}_n\ \vec{d}^{k,max})$ *qui contiendra les données du tableau* A_k *produites par des* $\vec{l'}$ *et qui seront consommées par des* $\vec{l}''$ *telles que* $l'_1 = l_1, \cdots, l'_{n-1} = l_{n-1}$, $l'_n = l_n - 1$, $l''_1 = l_1, \cdots, l''_n = l_n$;
– *un tampon* $B'_{k,n}$ *de taille* $\prod_{i=1}^{n-1}(\mathcal{E}_i\ \vec{T}) * (\mathcal{E}_n\ \vec{d}^{k,max})$ *qui contiendra les données du tableau* A_k *produites par des* $\vec{l'}$ *et qui seront consommées par des* $\vec{l}''$ *telles que* $l'_1 = l_1, \cdots, l'_n = l_n$, $l''_1 = l_1, \cdots, l''_{n-1} = l_{n-1}$ *et* $l''_n = l_n + 1$;
– *un tampon* $B_{k,n+1}$ *de taille* $(\mathcal{R}_1\ \vec{d}^{k,max} + 1) * \prod_{k=2}^{n}(\mathcal{R}_k \vec{T})$ *qui contiendra les données du tableau* A_k *produites par des* $\vec{l'}$ *et qui seront consommées par des* $\vec{l}''$ *telles que* $l'_1 = l_1, \cdots, l'_n = l_n$, $l''_1 = l_1, \cdots$ *et* $l''_n = l_n$.

avec :

1. $\vec{d}^{k,max}$ *est le vecteur des plus grandes projections de toutes les dépendances de l'instruction* S_k *vers l'instruction* S_{k+1} *du code en* figure 4.4 *(*page 36*)* ;
2. $\vec{T}$ *est le vecteur des tailles des tuiles* ;
3. $\vec{N}^k$ *est le vecteur du nombre d'itérations de chaque boucle du nid* k *du code en* figure 4.1 *(*page 32*)* ;
4. $\mathcal{E}_i$ *est la* $i^{\text{ème}}$ *ligne de la matrice* $\mathcal{E}$ *de permutation des boucles externes* ;
5. $\mathcal{R}_i$ *est la* $i^{\text{ème}}$ *ligne de la matrice* $\mathcal{R}$ *de permutation des boucles internes.*

Preuve : La preuve se fait de la même manière que le cas à deux dimensions. Cela revient à partitionner l'ensemble des données vivantes en plusieurs ensembles. Chacun de ces derniers va correspondre à l'un des tampons ou à une paire de tampons. □

4.3.6 Génération du code dans le cas général :

De la même manière que dans le cas à deux dimensions, pour éviter la copie du tampon $B'_{k,i}$ $(1 \leq i \leq n)$ dans le tampon $B_{k,i}$, nous fusionnons les tampons :

– $B_{k,1}$ et $B'_{k,1}$ en un seul tampon qui sera noté par $BT_{k,1}[2]$ tel que $|BT_{k,1}[0]| = |BT_{k,1}[1]| = |B_{k,1}| = |B'_{k,1}|$. Dans ce cas, une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,1}$, elle va la stocker dans $BT_{k,1}[l_1 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,1}$, elle va la récupérer dans $BT_{k,1}[(l_1 - 1) \bmod 2]$; ⋮
– $B_{k,m}$ et $B'_{k,m}$ en un seul tampon qui sera noté par $BT_{k,m}[2]$ tel que $|BT_{k,m}[0]| = |BT_{k,m}[1]| = |B_{k,m}| = |B'_{k,m}|$. Dans ce cas une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,m}$, elle va la stocker dans $BT_{k,m}[l_m \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,m}$, elle va la récupérer dans $BT_{k,m}[(l_m - 1) \bmod 2]$; ⋮

– $B_{k,n}$ et $B'_{k,n}$ en un seul tampon qui sera noté par $BT_{k,n}[2]$ tel que $|BT_{k,n}[0]| = |BT_{k,n}[1]| = |B_{k,n}| = |B'_{k,n}|$. Dans ce cas une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,n}$, elle va la stocker dans $BT_{k,n}[l_n \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,n}$, elle va la récupérer dans $BT_{k,n}[(l_n - 1) \bmod 2]$.

4.4 Conclusion

Un nombre important de techniques d'optimisations de la localité ont été proposées dans le passé [75, 80, 87, 80, 5]. La plupart de ces techniques utilisent les transformations unimodulaires et le pavage, qui sont appliqués uniquement à des codes qui sont constitués d'un seul nid de boucles parfaitement imbriquées. Les transformations unimodulaires sont utilisées pour rendre le nid de boucles complètement permutable, afin de pouvoir appliquer facilement un pavage parallèle aux axes.

Le peu de travaux sur le pavage qui sont appliqués à des nids de boucles mal imbriquées proposent d'appliquer des transformations supplémentaires comme la *fusion de boucles*, *fission* et *sinking* qui permettent de créer un nid de boucles parfaitement imbriquées et ensuite appliquer le pavage sur ce dernier [18, 50, 80, 75]. Ahmed [3, 2] applique le pavage sur un seul nid de boucles mal imbriquées. L'idée principale de cette technique est de transformer l'espace d'itération de chaque instruction dans le nid de boucles mal imbriquées en un espace spécial appelé *espace produit* qui est le produit cartésien des espaces d'itérations de toutes les instructions du nid de boucles mal imbriquées.

Kodura et al [51] proposent une autre technique appelée *data-shackling* qui consiste à diviser les tableaux de données en plusieurs blocs et ensuite choisir l'ordre dans lequel ces blocs seront chargés dans le cache. Leur algorithme ne spécifie pas comment se fait le découpage des tableaux en blocs. Pugh et el [62] ont proposé une autre technique appelée *iteration space slicing* qui donne presque les mêmes résultats que la fusion de boucles mais dans cette technique ils ne s'intéressent pas au pavage.

Lim et al [54] propose une technique de partitionnement qui prend en considération la contraction des tableaux [32, 67] et le pavage. Leur technique est appliquée à un seul nid de boucles de structure arbitraire. Leur idée consiste à trouver le meilleur partitionnement affine qui sera l'équivalent d'une combinaison des transformations unimodulaires, la fusion, fission et le réordonnancement des instructions.

Dans ce chapitre, nous avons appliqué simultanément le pavage, la fusion avec alignement et la permutation de boucles sur une séquence de nids de boucles parallèles qui n'ont pas nécessairement le même domaine d'itération. Notre méthode consiste à décaler les domaines d'itérations de tous les nids afin que l'application de notre pavage avec fusion soit valide. Grâce à ces décalages et à nos restrictions sur le code en entrée, nous pouvons toujours appliquer notre pavage avec fusion, ce qui n'est pas le cas dans les autres travaux. Grâce à la matrice de permutation $\mathcal{P}$ nous avons pris en considération tous les ordonnancements possibles des tuiles et des itérations à l'intérieur des ces dernières. La modélisation de notre pavage avec fusion comme une transformation de boucles nous a facilement permis de mettre en système d'inéquations les différents paramètres : (1) la matrice de permutation $\mathcal{P}$, (2) la matrice des tailles des tuiles $\mathcal{A}$, (3) les patrons des différents nids et (4) les vecteurs de décalages. À ce système d'inéquations nous avons donné deux types de solutions.

À notre connaissance, l'allocation des tampons dans le cas du pavage n'a été traitée par aucun auteur. Dans ce chapitre, nous avons proposé deux méthodes d'allocation des tampons. Le but de ces allocations est de réduire le nombre de données mortes qui seront stockées dans le cache. La première méthode consiste à remplacer chaque tableau intermédiaire par un tampon circulaire et séquentiel. Cette méthode possède deux inconvénients : (1) à certaines itérations le rapport entre les données mortes et les données vivantes qui sont stockées dans ce tampon est considérablement important, car la plus ancienne des productions varie en fonction de la position de l'itération dans la tuile (*figure 4.5*) et (2) ses fonctions d'adressages sont compliquées. Pour pallier les inconvénients de cette méthode, nous avons proposé dans la deuxième méthode de diviser le volume mémoire des données vivantes de chaque tableau en plusieurs sous-ensembles et d'attribuer à chacun de ces dernier un tampon. Pour ces différents tampons, nous avons défini des fonctions d'accès qui permettent de ranger et de récupérer les données du tableau correspondant. Cette technique est implantée dans $PIPS$ et nous verrons dans la *chapitre 7* qu'elle permet de réduire significativement les défauts de cache.

Chapitre 5

Fusion avec pavage à deux niveaux d'une séquence de nids de boucles

Après avoir traité, dans le chapitre précédent, la fusion avec pavage à un seul niveau d'une séquence de nids de boucles, nous traitons dans ce chapitre la fusion avec pavage à deux niveaux. Le premier niveau est destiné au cache et le deuxième aux registres. Nous transformons chaque nid de profondeur n en un autre nid de profondeur $3n$. Cela revient à partitionner l'espace d'itération en un ensemble de blocs que nous appelons les tuiles externes et chacune de ces tuiles externes sera partitionnée à son tour en un ensemble de sous-blocs qui seront appelés les tuiles internes. Les n boucles externes du nid transformé vont parcourir les tuiles externes, les n boucles intermédiaires les tuiles internes et les n boucles internes les itérations à l'intérieur des tuiles internes.

De la même manière que la fusion avec pavage à un seul niveau, notre pavage sera modélisé par deux matrices : (1) une matrice de pavage hiérarchique $\mathcal{A}(n,\ 3n)$ qui donne les tailles des tuiles externes et internes et (2) une matrice $\mathcal{P}(3n,\ 3n)$ qui permet de spécifier l'ordonnancement des différentes tuiles et des itérations à l'intérieur des tuiles internes.

5.1 Code en entrée

Les codes en entrée ont la même forme que ceux des *chapitres* 2 et 3, c'est-à-dire qu'ils ont la forme du code en *figure 5.1.*

Les contraintes sur ces codes sont toujours les mêmes, c'est-à-dire que :

1. tous les nids ont la même profondeur, qui est quelconque ;
2. les différents tableaux ont la même dimension, égale à la profondeur des nids ;
3. toutes les boucles sont parallèles ;
4. les bornes de ces différents nids sont des constantes numériques ;
5. les vecteurs des différents patrons sont ordonnés de la manière suivante :
$$\vec{v}^{k,1} \preceq \vec{v}^{k,2} \preceq \ldots.. \preceq \vec{v}^{k,m_k}.$$
6. le domaine D_0 est imposé par l'utilisateur et les autres domaines sont dérivés de la manière suivante :
$$D_k = \{\vec{i} \mid \forall \vec{v} \in R^k :\ \vec{i} + \vec{v} \in D_{k-1}\}$$

Comme nous l'avons mentionné dans le *chapitre 3* sur la fusion, cette dérivation permet d'éviter d'utiliser des éléments qui ne sont pas définis.

```
do $\vec{i}_1 \in D_1$
    $A_1(\vec{i}_1) = A_0(\vec{i}_1 + \vec{v}^1_1) \otimes .... \otimes A_0(\vec{i}_1 + \vec{v}^1_{m_1})$        (* $s_1$ *)
enddo
.
.
do $\vec{i}_k \in D_k$
    $A_k(\vec{i}_k) = A_{k-1}(\vec{i}_k + \vec{v}^k_1) \otimes .... \otimes A_{k-1}(\vec{i}_k + \vec{v}^k_{m_k})$   (* $s_k$ *)
enddo
.
.
do $\vec{i}_p \in D_p$
    $A_p(\vec{i}_p) = A_{p-1}(\vec{i}_p + \vec{v}^p_1) \otimes .... \otimes A_{p-1}(\vec{i}_p + \vec{v}^p_{m_p})$   (* $s_p$ *)
enddo
```

Figure 5.1: Code en entrée (séquence de nids de boucles). $\forall j : \vec{i}_j \in \mathbb{Z}^n$. La dimension des tableaux A_k est n et $\otimes$ représente une opération quelconque

5.2 Fusion avec pavage

De la même manière que les chapitres précédents, la première étape qui précède l'application de la fusion avec pavage est la translation de chaque domaine D_k par un vecteur $\vec{h}_k$. Dans la *section 4.2.5*, nous donnons la condition que doivent vérifier ces différents vecteurs pour que l'application de notre fusion avec pavage soit valide.

5.2.1 Translation des domaines des différents nids

Après la translation des différents domaines par des délais $\vec{h}_k$, nous obtenons le code en *figure 5.2*. La translation de chaque domaine D_k par le vecteur $\vec{h}_k$ sera notée par D'_k avec :

$$D'_k = \{\vec{i} = \vec{i}_k + \vec{h}_k \ |\vec{i}_k \in D_k\}$$

Durant tout le reste de ce chapitre, nous notons par :

1. $D' = \cup_{k=1}^{p} D'_k$;
2. $\vec{O} = \min_{<}(\vec{i} \in D')$;
3. $\vec{N} = \max_{<}(\vec{i} \in D')$.

Les vecteurs $\vec{O}$ et $\vec{N}$ ne sont pas nécessairement dans le domaine D' ; nous prenons la plus petite enveloppe convexe qui contiendra $\vec{O}$ et $\vec{N}$. Pour faciliter la génération du code, après le calcul des vecteurs de translation $\vec{h}_k$ qui rendront l'application du pavage avec fusion valide, nous translatons le domaine D' avec un vecteur $\vec{h}$ de façon à assurer que $\vec{O} = \vec{0}$. Cette dernière étape ne modifie pas la condition de légalité du pavage avec fusion, car tous les domaines seront décalés avec le même vecteur.

5.2.2 Matrice du pavage A

La matrice du pavage $\mathcal{A}(n,\ 3n)$ permet de spécifier les tailles des tuiles externes et internes. Cette matrice a la forme suivante :

do $\vec{i} \in D'_1$
$\quad A_1(\vec{i} - \vec{h}_1) = A_0(\vec{i} - \vec{h}_1 + \vec{v}^1_1) \otimes .... \otimes A_0(\vec{i} - \vec{h}_1 + \vec{v}^1_{m_1})$ (* s_1 *)
enddo
.
.
do $\vec{i} \in D'_k$
$\quad A_k(\vec{i} - \vec{h}_k) = A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_1) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{h}_k + \vec{v}^k_{m_k})$ (* s_k *)
enddo
.
.
do $\vec{i} \in D'_p$
$\quad A_p(\vec{i}_p - \vec{h}_p) = A_{p-1}(\vec{i}_p - \vec{h}_p + \vec{v}^p_1) \otimes .... \otimes A_{p-1}(\vec{i}_p - \vec{h}_p + \vec{v}^p_{m_p})$ (* s_p *)
enddo

Figure 5.2: Séquence de nids de boucles après la translation des différents nids

$$\mathcal{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{i,3i-2} & a_{i,3i-1} & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & a_{n,3n-2} & a_{n,3n-1} & 1 \end{pmatrix}$$

Tous les éléments de la $i^{\text{ème}}$ ligne de cette matrice sont nuls sauf :

1. $a_{i,3i-2}$ qui représente la taille des tuiles externes sur le $i^{\text{ème}}$ axe ;
2. $a_{i,3i-1}$ qui représente la taille des tuiles internes sur le $i^{\text{ème}}$ axe ;
3. $a_{i,3i}$ qui est égal à 1.

À partir de la matrice $\mathcal{A}$, nous pouvons définir une transformation ω_1 qui permet de transformer chaque point $\vec{i} = (i_1, \cdots, i_n)^t \in D' \subset \mathbb{Z}^n$ (*figure 5.2* en *page 69*) en un autre point $\vec{i'} = (i'_1, \cdots, i'_{3n})^t \in \omega_1(D') \subset \mathbb{Z}^{3n}$ (*figure 5.3* en *page 71*) :

$$\begin{array}{rcl} D' & \longrightarrow & \omega_1(D') \subset \mathbb{Z}^{3n} \\ \omega_1 : \vec{i} & \longmapsto & \vec{i'} = (\lfloor \frac{i_1}{a_{1,1}} \rfloor,\ \lfloor \frac{i_1 \bmod a_{1,1}}{a_{1,2}} \rfloor,\ i_1 \bmod a_{1,2}, \cdots, \lfloor \frac{i_m}{a_{m,3m-2}} \rfloor,\ \lfloor \frac{i_m \bmod a_{m,3m-2}}{a_{m,3m-1}} \rfloor, \\ & & i_m \bmod a_{m,3m-1}, \cdots, \lfloor \frac{i_n}{a_{n,3n-2}} \rfloor,\ \lfloor \frac{i_n \bmod a_{n,3n-2}}{a_{n,3n-1}} \rfloor,\ i_n \bmod a_{n,3n-1})^t \end{array}$$

Afin de faciliter la génération du code, nous supposons que $\vec{O} = \vec{0}$. Nous montrons dans la *section 4.2.7*, comment nous pouvons satisfaire cette condition.

Définition 27 *Soient $\vec{T^e}$ et $\vec{T^r}$ les vecteurs respectivement des tailles des tuiles externes et des tailles des tuiles internes ; $T^e_i = a_{i,3i-2}$ et $T^r_i = a_{i,3i-1}$. À partir du vecteur $\vec{i'}$, nous pouvons extraire trois sous-vecteurs :*

1. *un vecteur* $\vec{i'}_{\mathcal{E}} = \lfloor \frac{\vec{i}}{\vec{T^e}} \rfloor = (\lfloor \frac{i_1}{a_{1,1}} \rfloor, \cdots, \lfloor \frac{i_m}{a_{m,3m-2}} \rfloor, \cdots, \lfloor \frac{i_n}{a_{n,3n-2}} \rfloor)^t$ *qui parcourt les tuiles externes ;*
2. *un vecteur* $\vec{i'}_{\mathcal{R}} = \lfloor \frac{\vec{i} \bmod \vec{T^e}}{\vec{T^r}} \rfloor = (\lfloor \frac{i_1 \bmod a_{1,1}}{a_{1,2}} \rfloor, \cdots, \lfloor \frac{i_m \bmod a_{m,3m-2}}{a_{m,3m-1}} \rfloor, \cdots, \lfloor \frac{i_n \bmod a_{n,3n-2}}{a_{n,3n-1}} \rfloor)^t$ *qui parcourt les tuiles internes ;*
3. *un vecteur* $\vec{i'}_{\mathcal{S}} = \vec{i} \bmod \vec{T^r} = (i_1 \bmod a_{1,2}, \cdots, i_m \bmod a_{m,3m-1}, \cdots, i_n \bmod a_{n,3n-1})^t$ *qui parcourt les itérations à l'intérieur des tuiles internes.*

Propriété 2 *Nous pouvons écrire* $\vec{i'}$ *en fonction de* $\vec{i'}_{\mathcal{E}}$, $\vec{i'}_{\mathcal{R}}$ *et* $\vec{i}_{\mathcal{S}}$ *de la manière suivante :*

$$\vec{i'} = \mathcal{M} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix}, \text{avec :}$$

$\mathcal{M}$ *est une matrice de taille* $(3n,\ 3n)$ *que nous pouvons décomposer en trois sous-matrices. C'est-à-dire que* $\mathcal{M} = \big(\mathcal{M}_1\mathcal{M}_2\mathcal{M}_3\big)$, *telles que :*

$$\mathcal{M}_1(3n,\ n) = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \mathcal{M}_2(3n,\ n) = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{et } \mathcal{M}_3(3n,\ n) = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

La transformation ω_1 est bijective [84] et nous avons $\vec{i} = \mathcal{A}\ \vec{i'}$. Après l'application de la matrice $\mathcal{A}$ sur le code en *figure 5.2* (*page 69*), nous obtenons le code en *figure 5.3* (*page 71*).

Comme l'instruction S_k n'est pas exécutée pour toute itération du domaine $\omega_1(D')$ du code en *figure 5.3* (*page 71*), elle doit être gardée par la condition :

$$C_k(\vec{i'}) = \text{if } (\vec{i'} \in \omega_1(D'_k))$$

Dans le code tuilé, les boucles qui parcourent les tuiles externes doivent être placées avant les boucles qui parcourent les tuiles internes et ces dernières doivent être à leur tour

$$
\begin{array}{ll}
\mathsf{do}(\vec{i'} \in \omega_1(D')) & \\
\quad C_1(\vec{i'}) :\; A_1(\mathcal{A}\vec{i'} - \vec{h}_1) = A_0(\mathcal{A}\vec{i'} - \vec{h}_1 + \vec{v}^1_1) \otimes \cdots \otimes A_0(\mathcal{A}\vec{i'} - \vec{h}_1 + \vec{v}^1_{m_1}) & (*\ s_1\ *) \\
\quad . & \\
\quad . & \\
\quad C_k(\vec{i'}) :\; A_k(\mathcal{A}\vec{i'} - \vec{h}_k) = A_{k-1}(\mathcal{A}\vec{i'} - \vec{h}_k + \vec{v}^k_1) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\vec{i'} - \vec{h}_k + \vec{v}^k_{m_k}) & (*\ s_k\ *) \\
\quad . & \\
\quad . & \\
\quad C_p(\vec{i'})\; A_p(\mathcal{A}\vec{i'} - \vec{h}_p) = A_{p-1}(\mathcal{A}\vec{i'} - \vec{h}_p + \vec{v}^p_1) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\vec{i'} - \vec{h}_p + \vec{v}^p_{m_p}) & (*\ s_p\ *) \\
\mathsf{enddo} &
\end{array}
$$

Figure 5.3: Code après application de $\mathcal{A}$ et translation du domaine d'itération de chaque nid avec un délai $\vec{h}_k$

placées avant les boucles qui parcourent les itérations dans ces tuiles internes. Dans le code en *figure 5.3* (*page 71*), cette condition n'est pas vérifiée. Par conséquent, nous appliquons une matrice de permutation $\mathcal{P}$ sur ce code afin d'échanger plusieurs boucles en même temps.

5.2.3 Échange de boucles

Grâce à la matrice de permutation $\mathcal{P}(3n,\ 3n)$ qui permet d'échanger plusieurs boucles en même temps, nous pouvons spécifier l'ordre dans lequel les tuiles externes, les tuiles internes et les itérations à l'intérieur de ces dernières seront exécutées. Cette matrice aura la forme suivante :

$$
\begin{pmatrix}
p_{1,1} & \cdots & p_{1,j} & \cdots & p_{1,3n} \\
\vdots & \ddots & \vdots & & \vdots \\
p_{i,1} & \cdots & p_{i,j} & \cdots & p_{i,3n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
p_{3n,1} & \cdots & p_{3n,j} & \cdots & p_{3n,3n}
\end{pmatrix}
$$

Dans chaque ligne et dans chaque colonne de cette matrice, il y a un et un seul élément qui est égal à un ; les autres éléments sont tous nuls. C'est-à-dire que :

$$
\begin{array}{c}
\forall i,\ \exists! j :\ \mathcal{P}_{i,j} = 1 \\
\forall j,\ \exists! i :\ \mathcal{P}_{i,j} = 1
\end{array}
$$

La matrice $\mathcal{P}$ transforme chaque point $\vec{i'} \in \mathbb{Z}^{3n}$ du code en *figure 5.3* (*page 71*) en un autre point $\vec{l} \in \mathbb{Z}^{3n}$ du code en *figure 5.4* (*page 72*).

L'instruction S_k doit être gardée par :

$$C_k(\vec{l}) = \text{if } [\ \vec{l} \in \mathcal{P}\ (\omega_1(D'_k))\ ].$$

La relation entre $\vec{i'}$ et $\vec{l}$ est donnée par :

$$
\begin{array}{rcl}
\vec{l} & = & \mathcal{P}\ \vec{i'} \\
\vec{i'} & = & \mathcal{P}^{-1}\ \vec{l}
\end{array}
$$

do[$\vec{l} \in \mathcal{P}\ (\omega_1(D'))$]
$C_1(\vec{l}) :\ A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}^1_1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}^1_{m_1})$
.
.
$C_k(\vec{l}) :\ A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}^k_1) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}^k_{m_k})$
.
.
$C_p(\vec{l}) :\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}^p_1) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}^p_{m_p})$
enddo

Figure 5.4: Code après application de la matrice $\mathcal{P}$

5.2.4 Pavage contigu

Définition 28 (Pavage contigu) *Un pavage est dit contigu si et seulement si :*

1. $\forall k \in [1 \cdots n]\ , \exists j\ : l_k = \lfloor \frac{i_j}{a_{j,3j-2}} \rfloor$
2. $\forall k \in [n+1 \cdots 2n],\ \exists j\ : l_k = \lfloor \frac{i_j \mod a_{j,3j-2}}{a_{j,3j-1}} \rfloor$
3. $\forall k \in [2n+1 \cdots 3n],\ \exists j\ : l_k = (i_j \mod a_{j,3j-1})$

Cette définition signifie que toutes les boucles qui parcourent les tuiles externes doivent être placées avant les boucles qui parcourent les tuiles internes et les boucles qui parcourent ces tuiles internes doivent être placées avant les boucles qui parcourent les itérations dans ces dernières.

Certaines configurations de la matrice $\mathcal{P}$ ne sont pas admises, car elles ne donnent pas un pavage contigu. La *Proposition 16* donne les conditions que doit vérifier la matrice $\mathcal{P}$ pour que notre pavage soit contigu.

Proposition 16 *Un pavage est contigu si et seulement si :*

$$\begin{cases} \forall k\ :\ [(\exists j\ :\ \mathcal{P}_{k,3j-2} = 1) \Rightarrow 1 \le k \le n] \\ \forall k\ :\ [(\ \exists j\ :\ \mathcal{P}_{k,3j-1} = 1) \Rightarrow n+1 \le k \le 2n] \\ \forall k\ :\ [(\ \exists j\ :\ \mathcal{P}_{k,3j} = 1) \Rightarrow 2n+1 \le k \le 3n] \end{cases}$$

Preuve :

$\Rightarrow$? Nous supposons que :

– (a) notre pavage est contigu et

– (b) $\begin{cases} \exists k\ :\ [(\exists j\ :\ \mathcal{P}_{k,3j-2} = 1) \wedge k \ge n+1] \\ \text{ou} \\ \exists k\ :\ [(\ \exists j\ :\ \mathcal{P}_{k,3j-1} = 1) \wedge (k \le n \text{ ou } k \ge 2n+1)] \\ \text{ou} \\ \exists k\ :\ [(\ \exists j\ :\ \mathcal{P}_{k,3j} = 1) \wedge k \le 2n] \end{cases}$

1. $(\mathcal{P}_{k,3j-2} = 1) \wedge (k \ge n+1)$. Cela signifie que :

$$l_k = \lfloor \frac{i_j}{a_{j,3j-2}} \rfloor \wedge (k \ge n+1)$$

Contradiction avec l'hypothèse (a) (*définition 28* en *page 72*).

2. $(\mathcal{P}_{k,3j-1} = 1) \wedge (k \leq n$ ou $k \geq 2n+1)$. Cela signifie que :

$$l_k = \lfloor \frac{i_j \mod a_{j,3j-2}}{a_{j,3j-1}} \rfloor \wedge (k \leq n \text{ ou } k \geq 2n+1)$$

Contradiction avec l'hypothèse (a) (*définition 28* en *page 72*).

3. $(\mathcal{P}_{k,3j} = 1) \wedge (k \leq 2n)$. Cela signifie que :

$$l_k = (i_j \mod a_{j,3j-1}) \wedge (k \leq 2n)$$

Contradiction avec l'hypothèse (a) (*définition 28* en *page 72*).

$\Leftarrow$? Nous supposons que :

– (a) $\begin{cases} \forall k \ : \ [(\exists j \ : \ \mathcal{P}_{k,3j-2} = 1) \Rightarrow k \leq n] \\ \forall k \ : \ [(\exists j \ : \ \mathcal{P}_{k,3j-1} = 1) \Rightarrow n+1 \leq k \leq 2n] \\ \forall k \ : \ [(\exists j \ : \ \mathcal{P}_{k,3j} = 1) \Rightarrow 2n+1 \leq k \leq 3n] \end{cases}$

– (b) notre pavage n'est pas contigu.

D'après la *définition 28* (*page 72*), le pavage peut ne pas être contigu dans trois cas :

Cas 1 : $\exists k \in [1 \cdots n], \forall j \ : l_k \neq \lfloor \frac{i_j}{a_{j,3j-2}} \rfloor$. Cela signifie que :

$$\exists k \in [1 \cdots n], \exists j \ : l_k = \lfloor \frac{i_j \bmod a_{j,3j-2}}{a_{j,3j-1}} \rfloor \text{ ou } l_k = (i_j \bmod a_{j,3j-1})$$

Par conséquent :

$$\exists k \in [1 \cdots n], \exists j : \mathcal{P}_{k,3j-1} = 1 \text{ ou } \mathcal{P}_{k,3j} = 1$$

Contradiction avec l'hypothèse (a).

Cas 2 : $\exists k \in [n+1 \cdots 2n], \forall j \ : l_k \neq \lfloor \frac{i_j \mod a_{j,3j-2}}{a_{j,3j-1}} \rfloor$. Cela signifie que :

$$\exists k \in [n+1 \cdots 2n], \exists j \ : \ l_k = \lfloor \frac{i_j}{a_{j,3j-2}} \rfloor \text{ ou } l_k = (i_j \bmod a_{j,3j-1})$$

Par conséquent :

$$\exists k \in [n+1 \cdots 2n], \exists j \ : \mathcal{P}_{k,3j-2} = 1 \text{ ou } \mathcal{P}_{k,3j} = 1$$

Contradiction avec l'hypothèse (a).

Cas 3 : $\exists k \in [2n+1 \cdots 3n], \forall j \ : l_k \neq (i_j \bmod a_{j,3j-1})$. Cela signifie que :

$$\exists k \in [2n+1 \cdots 3n], \exists j \ : \ l_k = \lfloor \frac{i_j}{a_{j,3j-2}} \rfloor \text{ ou } l_k = \lfloor \frac{i_j \bmod a_{j,3j-2}}{a_{j,3j-1}} \rfloor$$

Par conséquent :

$$\exists k \in [2n+1 \cdots 3n], \exists j \ : \mathcal{P}_{k,3j-2} = 1 \text{ ou } \mathcal{P}_{k,3j-1} = 1$$

Contradiction avec l'hypothèse (a).

□

Tout au long de ce chapitre, nous considérons que des matrices $\mathcal{P}(3n,\ 3n)$ qui donnent un pavage contigu.

5.2.5 Décomposition de la matrice $\mathcal{P}$

Définition 29 *À partir de la matrice $\mathcal{P}(3n,\ 3n)$, nous pouvons extraire trois sous-matrices :*

1. *une matrice $\mathcal{E}(n,\ n)$ de permutation des boucles $\vec{i'}_{\mathcal{E}}$: $\mathcal{E}_{i,j} = \mathcal{P}_{i,3j-2}$*
2. *une matrice $\mathcal{R}(n,\ n)$ de permutation des boucles $\vec{i'}_{\mathcal{R}}$: $\mathcal{R}_{i,j} = \mathcal{P}_{i+n,3j-1}$*
3. *une matrice $\mathcal{S}(n,\ n)$ de permutation des boucles $\vec{i'}_{\mathcal{S}}$: $\mathcal{S}_{i,j} = \mathcal{P}_{i+2n,3j}$*

Proposition 17 *Si la matrice $\mathcal{P}$ satisfait les conditions données dans la* proposition 16 *(*page 72*), alors :*

$$\vec{l} = \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix}$$

Preuve : D'après la *propriété 2* (*page 70*) :

$$\vec{i'} = \mathcal{M} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix},\ \text{avec :}$$

$\mathcal{M}$ est une matrice de taille $(3n,\ 3n)$ que nous pouvons décomposer en trois sous-matrices : $\mathcal{M} = \begin{pmatrix} \mathcal{M}_1 & \mathcal{M}_2 & \mathcal{M}_3 \end{pmatrix}$.

Comme $\vec{l} = \mathcal{P}\ \vec{i'}$, alors :

$$\vec{l} = \mathcal{P}\ \begin{pmatrix} \mathcal{M}_1 & \mathcal{M}_2 & \mathcal{M}_3 \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix}$$

Nous pouvons vérifier facilement que $\mathcal{P}\ \begin{pmatrix} \mathcal{M}_1 & \mathcal{M}_2 & \mathcal{M}_3 \end{pmatrix} = \begin{pmatrix} \mathcal{Q}_{1,1} & \mathcal{Q}_{1,2} & \mathcal{Q}_{1,3} \\ \mathcal{Q}_{2,1} & \mathcal{Q}_{2,2} & \mathcal{Q}_{2,3} \\ \mathcal{Q}_{3,1} & \mathcal{Q}_{3,2} & \mathcal{Q}_{3,3} \end{pmatrix}$, avec :

$$\mathcal{Q}_{1,1} = \begin{pmatrix} p_{1,1} & p_{1,4} & \cdots & p_{1,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{i,1} & p_{i,4} & \cdots & p_{i,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{n,1} & p_{n,4} & \cdots & p_{n,3n-2} \end{pmatrix} \mathcal{Q}_{1,2} = \begin{pmatrix} p_{1,2} & p_{1,5} & \cdots & p_{1,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{i,2} & p_{i,5} & \cdots & p_{i,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{n,2} & p_{n,5} & \cdots & p_{n,3n-1} \end{pmatrix}$$

$$\mathcal{Q}_{1,3} = \begin{pmatrix} p_{1,3} & p_{1,6} & \cdots & p_{1,3n} \\ \vdots & \vdots & & \vdots \\ p_{i,3} & p_{i,6} & \cdots & p_{i,3n} \\ \vdots & \vdots & & \vdots \\ p_{n,3} & p_{n,6} & \cdots & p_{n,3n} \end{pmatrix} \mathcal{Q}_{2,1} = \begin{pmatrix} p_{n+1,1} & p_{n+1,4} & \cdots & p_{n+1,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{n+i,1} & p_{n+i,4} & \cdots & p_{n+i,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{2n,1} & p_{2n,4} & \cdots & p_{2n,3n-2} \end{pmatrix}$$

$$\mathcal{Q}_{2,2} = \begin{pmatrix} p_{n+1,2} & p_{n+1,5} & \cdots & p_{n+1,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{n+i,2} & p_{n+i,5} & \cdots & p_{n+i,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{2n,2} & p_{2n,5} & \cdots & p_{2n,3n-1} \end{pmatrix} \mathcal{Q}_{2,3} = \begin{pmatrix} p_{n+1,3} & p_{n+1,6} & \cdots & p_{n+1,3n} \\ \vdots & \vdots & & \vdots \\ p_{n+i,3} & p_{n+i,6} & \cdots & p_{n+i,3n} \\ \vdots & \vdots & & \vdots \\ p_{2n,3} & p_{2n,6} & \cdots & p_{2n,3n} \end{pmatrix}$$

$$\mathcal{Q}_{3,1} = \begin{pmatrix} p_{2n+1,1} & p_{2n+1,4} & \cdots & p_{2n+1,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{2n+i,1} & p_{2n+i,4} & \cdots & p_{2n+i,3n-2} \\ \vdots & \vdots & & \vdots \\ p_{3n,1} & p_{3n,4} & \cdots & p_{3n,3n-2} \end{pmatrix} \mathcal{Q}_{3,2} = \begin{pmatrix} p_{2n+1,2} & p_{2n+1,5} & \cdots & p_{2n+1,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{2n+i,2} & p_{2n+i,5} & \cdots & p_{2n+i,3n-1} \\ \vdots & \vdots & & \vdots \\ p_{3n,2} & p_{3n,5} & \cdots & p_{3n,3n-1} \end{pmatrix}$$

$$\text{et } \mathcal{Q}_{3,3} = \begin{pmatrix} p_{2n+1,3} & p_{2n+1,6} & \cdots & p_{2n+1,3n} \\ \vdots & \vdots & & \vdots \\ p_{2n+i,3} & p_{2n+i,6} & \cdots & p_{2n+i,3n} \\ \vdots & \vdots & & \vdots \\ p_{3n,3} & p_{3n,6} & \cdots & p_{3n,3n} \end{pmatrix}$$

D'après la *définition 29* (*page 74*) :

$$\mathcal{E} = \mathcal{Q}_{1,1}, \mathcal{R} = \mathcal{Q}_{2,2} \text{ et } \mathcal{S} = \mathcal{Q}_{3,3}$$

D'après la *proposition 16* (*page 72*), les éléments des matrices $\mathcal{Q}_{1,2}$, $\mathcal{Q}_{1,3}$, $\mathcal{Q}_{2,1}$, $\mathcal{Q}_{2,3}$, $\mathcal{Q}_{3,1}$ et $\mathcal{Q}_{3,2}$ sont différents de 1. Par conséquent, ils sont nuls par définition d'une matrice de permutation $\mathcal{P}$.

Par conséquent :

$$\vec{l} = \begin{pmatrix} \mathcal{Q}_{1,1} & \mathcal{Q}_{1,2} & \mathcal{Q}_{1,3} \\ \mathcal{Q}_{2,1} & \mathcal{Q}_{2,2} & \mathcal{Q}_{2,3} \\ \mathcal{Q}_{3,1} & \mathcal{Q}_{3,2} & \mathcal{Q}_{3,3} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix} = \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix}$$

□

5.2.6 Modélisation du pavage

Notre pavage avec fusion sera représenté par une transformation de boucles ω_2 qui transforme chaque point $\vec{i} \in D' = \cup_{k=1}^{p} D'_k$ en un autre point $\vec{l} \in \mathbb{Z}^{3n}$.

$$\begin{array}{rcl} D' & \longrightarrow & \mathbb{Z}^{2n} \\ \omega_2 : \vec{i} & \longmapsto & \vec{l} = \mathcal{P} \,.\vec{i'} \\ & & \vec{l} = \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \vec{i'}_{\mathcal{E}} \\ \vec{i'}_{\mathcal{R}} \\ \vec{i'}_{\mathcal{S}} \end{pmatrix} \end{array} \tag{5.1}$$

5.2.7 Condition de légalité du pavage avec fusion

Comme pour la fusion avec pavage à un seul niveau, notre pavage avec fusion n'est correcte que si garantissons que toutes les données du tableau A_k qui sont nécessaires pour le calcul d'un élément du tableau A_{k+1} à une itération $\vec{l}$ dans le code tuilé et fusionné ont déjà été produites par les itérations précédentes.

Lemme 9 *Le pavage avec fusion défini par ω_2 est valide si et seulement si :*

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q : \omega_2(\vec{i} + \vec{v}_q^{k+1} - \vec{h}_{k+1} + \vec{h}_k) \preceq \omega_2(\vec{i}) \tag{5.2}$$

Preuve : Même preuve que dans le cas de la fusion avec pavage à un seul niveau (*page 40*).

Théorème 9 *Soient $\vec{d}_q^k = -\vec{v}_q^{k+1} + \vec{h}_{k+1} - \vec{h}_k$* la dépendance de flot *de l'instruction S_k vers l'instruction S_{k+1} du code en* figure 5.4 (page 72)*, $\vec{T}^e$ le vecteur des tailles des tuiles externes et $\vec{T}^r$ le vecteur des tailles des tuiles internes. Le pavage avec fusion défini par ω_2 est valide si et seulement si :*

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q : \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i} - \vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} \preceq \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{i} \bmod \vec{T}^r \end{pmatrix} \tag{5.3}$$

Preuve : En posant $\vec{j} = \vec{i} - \vec{d}_q^k$ ($\vec{d}_q^k = -\vec{v}_q^{k+1} + \vec{h}_{k+1} - \vec{h}_k$), le *lemme 9* sera équivalent à :

$$\forall k,\ \forall \vec{i} \in D'_{k+1}\ ,\ \forall q :\ \omega_2(\vec{j}) \preceq \omega_2(\vec{i})$$

D'après la définition de la transformation ω_2 (*relation 5.1* en *page 75*) :

$$\begin{aligned}
\omega_2(\vec{j}) &= \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \vec{j}'_{\mathcal{E}} \\ \vec{j}'_{\mathcal{R}} \\ \vec{j}'_{\mathcal{S}} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{j}}{\vec{T}^r} \rfloor \\ \lfloor \frac{\vec{j} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{j} \bmod \vec{T}^r \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}^r} \rfloor \\ \lfloor \frac{(\vec{i} - \vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} \\
\omega_2(\vec{i}) &= \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \vec{i}'_{\mathcal{E}} \\ \vec{i}'_{\mathcal{R}} \\ \vec{i}'_{\mathcal{S}} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^r} \rfloor \\ \lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{i} \bmod \vec{T}^r \end{pmatrix}
\end{aligned}$$

Par conséquent, le *lemme 9* sera équivalent à :

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q : \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i} - \vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} \preceq \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{i} \bmod \vec{T}^r \end{pmatrix}$$

□

5.2.8 Solutions possibles (Conditions suffisantes)

De la même manière que le chapitre précédent, nous donnons les deux solutions les plus importantes pour les délais $\vec{h}_k$ qui sont données dans les *théorèmes 10 et 12.*

Théorème 10 *Si* $\forall k\ ,\forall q\ :\ \vec{d}_q^k = -\vec{v}_q^{k+1} + \vec{h}_{k+1} - \vec{h}_k \geq \vec{0}$, *le pavage avec fusion défini par* ω_2 *est valide.*

Preuve : Comme $\vec{d}_q^k \geq \vec{0}$, alors nous avons deux cas :

– Cas 1 : $\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \leq \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$ et $\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \neq \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$.
  Comme $\mathcal{E}$ est une matrice de permutation alors :

$$\mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \leq \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor \text{ et } \mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \neq \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$$

  Nous savons que si $\vec{a} \leq \vec{b}$ et $\vec{a} \neq \vec{b}$ alors $\vec{a} \prec \vec{b}$. Par conséquent :

$$\mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \prec \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$$

  Comme nous avons :

  1. $\begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor\\ \lfloor\frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ (\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r\end{pmatrix} = \begin{pmatrix}\mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor\\ \mathcal{R}\lfloor\frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ \mathcal{S}[(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r]\end{pmatrix}$
  2. $\begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor\\ \lfloor\frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ \vec{i} \bmod \vec{T}^r\end{pmatrix} = \begin{pmatrix}\mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor\\ \mathcal{R}\lfloor\frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ \mathcal{S}[\vec{i} \bmod \vec{T}^r]\end{pmatrix}$
  3. $\mathcal{E}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \prec \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$

  Alors :

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q:\ \begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor\\ \lfloor\frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ (\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r\end{pmatrix} \prec \begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor\\ \lfloor\frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r}\rfloor\\ \vec{i} \bmod \vec{T}^r\end{pmatrix}$$

  D'après le *théorème 9* (*page 76*), le pavage avec fusion défini par ω_2 est valide.

– Cas 2 : $\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor = \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$.
  Nous savons que $\vec{a} \bmod \vec{b} = \vec{a} - \vec{b}\ \mathrm{o}\ \lfloor\frac{\vec{a}}{\vec{b}}\rfloor$ (*définition 9* en *page 8*). Par conséquent :

  1. $(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e = (\vec{i}-\vec{d}_q^k) - \vec{T}^e \mathrm{o}\ \lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor$
  2. $\vec{i} \bmod \vec{T}^e = \vec{i} - \vec{T}^e\ o\ \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$

  Comme $(\lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor = \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor)$ et $(\vec{d}_q^k \geq \vec{0})$ alors $(\vec{i}-\vec{d}_q^k) - \vec{T}^e\ o\ \lfloor\frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e}\rfloor \leq \vec{i} - \vec{T}^e\ o\ \lfloor\frac{\vec{i}}{\vec{T}^e}\rfloor$. Cela signifie que :

$$(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e \leq \vec{i} \bmod \vec{T}^e$$

Comme $\vec{T^r} \geq \vec{0}$, alors :

$$\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \leq \lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$$

Comme $\mathcal{R}$ est une matrice de permutation, alors :

$$\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \leq \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$$

Cette dernière relation nous donne deux cas :

- Cas a : $\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \leq \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$ et $\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \neq \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$. Cela signifie que :
  $\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \prec \mathcal{R}\lfloor\frac{(\vec{i}) \mod \vec{T^e}}{\vec{T^r}}\rfloor$
  Comme nous avons :
  1. $\begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}-\vec{d_q^k}}{\vec{T^e}}\rfloor\\ \lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ (\vec{i}-\vec{d_q^k}) \bmod \vec{T^r}\end{pmatrix} = \begin{pmatrix}\mathcal{E}\lfloor\frac{\vec{i}-\vec{d_q^k}}{\vec{T^e}}\rfloor\\ \mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ \mathcal{S}[(\vec{i}-\vec{d_q^k}) \bmod \vec{T^r}]\end{pmatrix}$
  2. $\begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}}{\vec{T^e}}\rfloor\\ \lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ \vec{i} \bmod \vec{T^r}\end{pmatrix} = \begin{pmatrix}\mathcal{E}\lfloor\frac{\vec{i}}{\vec{T^e}}\rfloor\\ \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ \mathcal{S}[\vec{i} \bmod \vec{T^r}]\end{pmatrix}$
  3. $\mathcal{E}\lfloor\frac{\vec{i}-\vec{d_q^k}}{\vec{T^e}}\rfloor = \mathcal{E}\lfloor\frac{\vec{i}}{\vec{T^e}}\rfloor$
  4. $\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor \prec \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$

  Alors

$$\forall k,\ \forall\vec{i} \in D'_{k+1},\ \forall q : \begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}-\vec{d_q^k}}{\vec{T^e}}\rfloor\\ \lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ (\vec{i}-\vec{d_q^k}) \bmod \vec{T^r}\end{pmatrix} \prec \begin{pmatrix}\mathcal{E} & 0 & 0\\ 0 & \mathcal{R} & 0\\ 0 & 0 & \mathcal{S}\end{pmatrix}\begin{pmatrix}\lfloor\frac{\vec{i}}{\vec{T^e}}\rfloor\\ \lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ \vec{i} \bmod \vec{T^r}\end{pmatrix}$$

  D'après le *théorème 9* (*page 76*), le pavage avec fusion défini par ω_2 est valide.
- Cas b : $\mathcal{R}\lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor = \mathcal{R}\lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor \Rightarrow \lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor = \lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor$.
  Comme $\vec{T^e} \bmod \vec{T^r} = \vec{0}$ alors :

$$\begin{aligned}\vec{i} \bmod \vec{T^r} &= (\vec{i} \bmod \vec{T^e}) \bmod \vec{T^r}\\ (\vec{i}-\vec{d_q^k}) \bmod \vec{T^r} &= ((\vec{i}-\vec{d_q^k}) \bmod \vec{T^e}) \bmod \vec{T^r}\end{aligned}$$

  Par définition de l'opérateur mod, nous avons :

$$\begin{aligned}(\vec{i} \bmod \vec{T^e}) \bmod \vec{T^r} &= (\vec{i} \bmod \vec{T^e}) - \vec{T^r}\ o\ \lfloor\frac{\vec{i} \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ ((\vec{i}-\vec{d_q^k}) \bmod \vec{T^e}) \bmod \vec{T^r} &= ((\vec{i}-\vec{d_q^k}) \bmod \vec{T^e}) - \vec{T^r}\ o\ \lfloor\frac{(\vec{i}-\vec{d_q^k}) \mod \vec{T^e}}{\vec{T^r}}\rfloor\\ (\vec{i} \bmod \vec{T^e}) &= \vec{i} - \vec{T^e}\ o\ \lfloor\frac{\vec{i}}{\vec{T^e}}\rfloor\\ (\vec{i}-\vec{d_q^k}) \bmod \vec{T^e} &= (\vec{i}-\vec{d_q^k}) - \vec{T^e}\ o\ \lfloor\frac{\vec{i}-\vec{d_q^k}}{\vec{T^e}}\rfloor\end{aligned}$$

Par conséquent :

$$\begin{aligned}
\vec{i} \bmod \vec{T}^r &= \vec{i} - \vec{T}^e\ o\ \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor - \vec{T}^r\ o\ \lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\
(\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r &= (\vec{i} - \vec{d}_q^k) - \vec{T}^e\ o\ \lfloor \frac{\vec{i} - \vec{d}_q^k}{\vec{T}^e} \rfloor - \vec{T}^r\ o\ \lfloor \frac{(\vec{i} - \vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor
\end{aligned}$$

Comme

1. $\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor = \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor$.
2. $\lfloor \frac{(\vec{i}-\vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor = \lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor$.
3. $\vec{d}_q^k \geq \vec{0}$

Alors :

$$\vec{i} \bmod \vec{T}^r \leq (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r$$

Comme $\mathcal{S}$ est une matrice de permutation alors :

$$\mathcal{S}(\vec{i} \bmod \vec{T}^r) \leq \mathcal{S}((\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r) \Rightarrow \mathcal{S}(\vec{i} \bmod \vec{T}^r) \preceq \mathcal{S}((\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r)$$

Comme nous avons :

1. $\begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i}-\vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \mathcal{R}\lfloor \frac{(\vec{i}-\vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ \mathcal{S}[(\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r] \end{pmatrix}$
2. $\begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{i} \bmod \vec{T}^r \end{pmatrix} = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \mathcal{R}\lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ \mathcal{S}[\vec{i} \bmod \vec{T}^r] \end{pmatrix}$
3. $\mathcal{E}\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor = \mathcal{E}\lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor$
4. $\mathcal{R}\lfloor \frac{(\vec{i}-\vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor = \mathcal{R}\lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor$
5. $\mathcal{S}(\vec{i} \bmod \vec{T}^r) \preceq \mathcal{S}((\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r)$

Alors :

$$\forall k,\ \forall \vec{i} \in D'_{k+1},\ \forall q : \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i}-\vec{d}_q^k)\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i} - \vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} \prec \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i}\ \bmod\ \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i}) \bmod \vec{T}^r \end{pmatrix}$$

D'après le *théorème 9* (*page 76*), le pavage avec fusion défini par ω_2 est valide.

□

Théorème 11 *Si* $\forall k\ :\ \vec{h}_k = \vec{h}_{k+1} - \max_{<}(v \in R^{k+1})$, *alors le pavage avec fusion défini par* ω_2 *est valide.*

Preuve : Même preuve que le *théorème 5* (*page 42*) dans le chapitre précédant.

□

Théorème 12 *Si* $\forall k, \forall q \; : \; \mathcal{E}\lfloor \frac{\vec{d}_q^k}{\vec{T}^e} \rfloor \succeq \vec{0}$*, alors le pavage avec fusion défini par* ω_2 *est valide.*

Preuve : De la même manière que la preuve du *théorème 6* (*page 43*) dans le chapitre précédent, nous obtenons deux cas :

- Cas 1 : $\vec{d}_q^k \geq \vec{0}$.
  D'après le *théorème 10* (*page 77*), le pavage avec fusion défini par ω_2 est valide.
- Cas 2 : $\mathcal{E}\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \prec \mathcal{E}\lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor$.
  Comme nous avons :

  1. $\begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \mathcal{R}\lfloor \frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \mathcal{S}[(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r] \end{pmatrix}$
  2. $\begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \vec{i} \bmod \vec{T}^r \end{pmatrix} = \begin{pmatrix} \mathcal{E}\lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \mathcal{R}\lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ \mathcal{S}[\vec{i} \bmod \vec{T}^r] \end{pmatrix}$
  3. $\mathcal{E}\lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \prec \mathcal{E}\lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor$

  Alors :

$$\forall k, \forall \vec{i} \in D'_{k+1}, \; \forall q : \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}-\vec{d}_q^k}{\vec{T}^e} \rfloor \\ \lfloor \frac{(\vec{i}-\vec{d}_q^k) \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i}-\vec{d}_q^k) \bmod \vec{T}^r \end{pmatrix} \prec \begin{pmatrix} \mathcal{E} & 0 & 0 \\ 0 & \mathcal{R} & 0 \\ 0 & 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} \lfloor \frac{\vec{i}}{\vec{T}^e} \rfloor \\ \lfloor \frac{\vec{i} \bmod \vec{T}^e}{\vec{T}^r} \rfloor \\ (\vec{i}) \bmod \vec{T}^r \end{pmatrix}$$

  D'après le *théorème 9* (*page 76*), le pavage avec fusion défini par ω_2 est valide.

□

La solution donnée dans ce dernier *théorème* n'est pas intéressante car la projection des dépendances sur certains axes sont très grandes lorsque les tuiles externes sont larges. Dans la suite de ce chapitre, nous considérons la solution donnée dans le *théorème 10* (*page 77*).

5.2.9 Les étapes de calcul des délais $\vec{h}_k$

Comme nous pouvons le remarquer dans le *théorème 10* (*page 77*), le calcul du délai $\vec{h}_{k+1}$ dépend du délai $\vec{h}_k$. Par conséquent, nous commençons à donner une valeur arbitraire au délai $\vec{h}_p$ (par exemple $\vec{0}$) et nous calculons successivement $\vec{h}_{p-1}$ en fonction de $\vec{h}_p$, $\vec{h}_{p-2}$ en fonction de $\vec{h}_{p-1}$... et enfin $\vec{h}_1$ en fonction de $\vec{h}_2$.

Une fois ces délais sont calculés, nous translatons les différents domaines D'_k par $-\vec{O} = -\min_{<}(\vec{i} \in D' = \cup_{m=1}^{p}(D'_m))$ afin d'assurer que $\vec{O} = \vec{0}$. Cela revient à soustraire de tous les délais $\vec{h}_k$ le vecteur $\vec{O}$. Cette dernière étape ne modifie pas la validité de notre pavage avec fusion, car tous les calculs seront décalés par la même valeur.

Dans la suite de ce chapitre, la translation du domaine D_k par $\vec{h}_k - \vec{O}$ sera notée par $D'_k = \{\vec{i} \; : \; \vec{O}_k \leq \vec{i} \leq \vec{N}_k + \vec{O}_k - \vec{1}\}$.

5.2.10 Conclusion

Grâce à la modélisation du pavage comme une transformation de boucles, nous avons mis en relation les différents paramètres et nous avons donné deux solutions aux vecteurs de translation $\vec{h}_k$ afin que notre pavage avec fusion soit valide. Maintenant nous proposons une méthode d'allocation des tampons et des registres qui permet d'augmenter davantage la localité des données.

5.3 Allocation des registres et des tampons

Dans ce chapitre, le tableau A_k sera remplacé par un ensemble de tampons et de registres. Notre méthode consiste toujours à décomposer l'ensemble des données vivantes du tableau A_k en plusieurs sous-ensembles : (1) l'un de ces sous-ensembles va correspondre aux registres et (2) chacun des autres va correspondre à un tampon.

5.3.1 Cas des nids de profondeur 2

Nous traitons d'abord le cas à deux dimensions et nous considérons que la matrice de permutation $\mathcal{P}$ est fixe :

1. Les différents nids du code en entrée donné en *figure 5.1* (*page 68*) sont de profondeur $n = 2$;
2. la matrice de permutation $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Comme la profondeur des différents nids est $n = 2$, la matrice de pavage hiérarchique $\mathcal{A}$ sera de taille $(n,\ 3n) = (2,\ 6)$ et elle aura la forme suivante :

$$\mathcal{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{2,4} & a_{2,5} & 1 \end{pmatrix} \text{ avec :}$$

1. $a_{1,1}$ et $a_{2,4}$ sont les tailles des tuiles externes respectivement sur l'axe i et l'axe j ;
2. $a_{1,2}$ et $a_{2,5}$ sont les tailles des tuiles internes respectivement sur l'axe i et l'axe j.

Après l'application de la fusion avec pavage définie par les matrice $\mathcal{A}$ et $\mathcal{P}$, nous obtenons le code en *figure 5.5* (*page 82*).

1. l_1 permet de parcourir les grandes tuiles sur l'axe i ;
2. l_2 permet de parcourir les grandes tuiles sur l'axe j ;
3. l_3 permet de parcourir les petites tuiles sur l'axe i ;
4. l_4 permet de parcourir les petites tuiles sur l'axe j ;
5. l_5 permet de parcourir les itérations à l'intérieur des petites tuiles sur l'axe i ;
6. l_6 permet de parcourir les itérations à l'intérieur des petites tuiles sur l'axe j.

L'ordre d'exécution des itérations est donné en *figure 5.6* (*page 82*), qui est le suivant :

1. Les grandes tuiles seront exécutées dans l'ordre *1*, *2*, *3*, *4*, *5* et *6*.
2. Les petites tuiles seront exécutées dans l'ordre a, b, c et d.
3. L'ordre d'exécution des itérations dans les petites tuiles est indiqué dans la tuile a.

```
do(l_1)
 do(l_2)
  do(l_3)
   do(l_4)
    do(l_5)
     do(l_6)
     C_1(l) : ...
     .
     .
     C_k(l) : ...
     .
     .
     C_p(l) : ...
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

do(l_1)
do(l_2)
do(l_3)
do(l_4)
do(l_5)
do(l_6)

$C_1(\vec{l}) : \; A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$

.
.

$C_k(\vec{l}) : \; A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)$

.
.

$C_p(\vec{l}) : \; A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)$

enddo
enddo
enddo
enddo
enddo
enddo

Figure 5.5: Code après application de la fusion et du pavage défini par $\mathcal{A}$ et $\mathcal{P}$

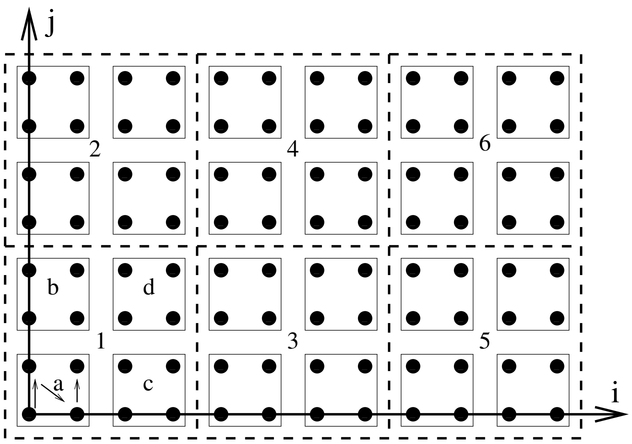


Figure 5.6: Domaine d'itération après l'application du pavage

5.3.2 Déroulage des boucles les plus internes dans le cas à deux dimensions

Chaque nouvelle itération du code en *figure 5.5* (*page 82*) produit un seul élément du tableau A_k. Nous transformons ce code en un autre qui produira plusieurs éléments en une seule itération. Chacun de ces éléments sera sauvegardé dans un registre. Pour cela, nous déroulons dans le code en *figure 5.5* entièrement les boucles qui parcourent les itérations à l'intérieur des petites tuiles [41, 40] ; cela revient à :

1. supprimer les boucles l_5 et l_6 ;
2. dupliquer les instructions S_1, S_2,$\cdots$ et S_p autant de fois que le nombre de valeurs que peut prendre le vecteur d'itération $(l_5,\ l_6)^t$.

Comme $0 \leq l_5 \leq T_1^r - 1 = a_{1,2} - 1$ et $0 \leq l_6 \leq T_2^r - 1 = a_{2,5} - 1$, alors après le déroulage de ces deux boucles, le code en *figure 5.5* (*page 82*) sera transformé en code de la *figure 5.7* (*page 84*) .

$$
\begin{array}{l}
\mathsf{do}(l_1)\\
\quad \mathsf{do}(l_2)\\
\quad\quad \mathsf{do}(l_3)\\
\quad\quad\quad \mathsf{do}(l_4)\\
\quad\quad\quad\quad \vec{l} = (l_1,\ l_2,\ l_3,\ l_4,\ 0,\ 0)^t\\
\quad\quad\quad\quad C_1(\vec{l}):\ A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_k(\vec{l}):\ A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_p(\vec{l}):\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad \vec{l} = (l_1,\ l_2,\ l_3,\ l_4,\ i,\ j)^t\\
\quad\quad\quad\quad C_1(\vec{l}):\ A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_k(\vec{l}):\ A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_p(\vec{l}):\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad \vec{l} = (l_1,\ l_2,\ l_3,\ l_4,\ T_1^r - 1,\ T_2^r - 1)^t\\
\quad\quad\quad\quad C_1(\vec{l}):\ A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_k(\vec{l}):\ A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)\\
\quad\quad\quad\quad .\\
\quad\quad\quad\quad C_p(\vec{l}):\ A_p(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)\\
\quad\quad\quad \mathsf{enddo}\\
\quad\quad \mathsf{enddo}\\
\quad \mathsf{enddo}\\
\mathsf{enddo}
\end{array}
$$

Figure 5.7: Code après le déroulage des boucles l_5 et l_6

5.3.3 Partition des données vivantes dans le cas à deux dimensions

Définition 30 *Soient* $(l_1,\ l_2,\ l_3,\ l_4)^t$ *une itération du code* figure 5.7 *(*page 84*) et* $(l_5,\ l_6)^t = (i,\ j)^t$*. Dans la suite de ce chapitre :*

1. $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ *sera appelée une itération du code en* figure 5.7 *;*
2. $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t$ *sera appelée une itération déroulée du code en* figure 5.7.

À chaque itération déroulée $\vec{l} \in \omega_2(D'_{k+1})$ du code en *figure 5.7* (*page 84*), nous calculons un nouvel élément du tableau A_{k+1} en fonction de m_{k+1} éléments du tableau A_k qui sont produits respectivement par les itérations déroulées

$$\vec{l}^1, \cdots, \vec{l}^{m_{k+1}}$$

Nous notons $\vec{d}^{k,q}$ la dépendance entre $\vec{i}^q = \mathcal{AP}^{-1}(\vec{l}^q)$ et $\vec{i} = \mathcal{AP}^{-1}(\vec{l})$. Il s'agit de la dépendance par rapport aux axes i et j. Au long de tout ce qui suit, nous supposons que $\forall m,\ \forall q\ : 0 \leq d_m^{k,q} \leq T_m^e = a_{m,3m-2}$, c'est-à-dire que la projection de n'importe quelle dépendance sur un axe donné est positive et ne dépasse pas la taille des tuiles externes sur cet axe.

Proposition 18 *Soit* $\vec{l'} \in \omega_2(D'_k)$ *une itération déroulée qui produit un élément du tableau* A_k *qui sera consommé par une itération déroulée* $\vec{l} \in \omega_2(D'_{k+1})$*. Comme* $\forall m,\ \forall q\ : 0 \leq d_m^{k,q} \leq T_m^e = a_{m,3m-2}$*, alors :*

1. $l'_1 \geq l_1 - 1$
2. $l'_2 \geq l_2 - 1$

Preuve : Soient $\vec{i'}$ et $\vec{i}$ les itérations dans l'espace initial qui correspondent aux itérations $\vec{l'}$ et $\vec{l}$. C'est-à-dire que $\vec{i} = \mathcal{AP}^{-1}\vec{l}$ et $\vec{i'} = \mathcal{AP}^{-1}\vec{l'}$. Nous savons que :

1. $\vec{i'} = \vec{i} - \vec{d}$ avec $\vec{d} \geq \vec{O}$;
2. $l_1 = \lfloor \frac{i_1}{a_{1,1}} \rfloor$;
3. $l_2 = \lfloor \frac{i_2}{a_{2,4}} \rfloor$;
4. $l'_1 = \lfloor \frac{i'_1}{a_{1,1}} \rfloor = \lfloor \frac{i_1 - d_1}{a_{1,1}} \rfloor$;
5. $l'_2 = \lfloor \frac{i'_2}{a_{2,4}} \rfloor = \lfloor \frac{i_2 - d_2}{a_{2,4}} \rfloor$.

Comme nous avons supposé que $0 \leq d_1 \leq a_{1,1}$ et $0 \leq d_2 \leq a_{2,4}$, alors :

1. $\lfloor \frac{i_1 - d_1}{a_{1,1}} \rfloor \geq \lfloor \frac{i_1}{a_{1,1}} \rfloor - 1 \Rightarrow l'_1 \geq l_1 - 1$;
2. $\lfloor \frac{i_2 - d_2}{a_{2,4}} \rfloor \geq \lfloor \frac{i_2}{a_{2,4}} \rfloor - 1 \Rightarrow l'_2 \geq l_2 - 1$.

Proposition 19 *Soit* $\vec{l'} \in \omega_2(D'_k)$ *une itération déroulée qui produit un élément du tableau* A_k *qui sera consommé par une itération déroulée* $\vec{l} \in \omega_2(D'_{k+1})$*. L'itération* $\vec{l'}$ *peut se trouver dans trois zones possibles :*

$$\begin{aligned}
Z_{k,1}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1 - 1\} ; \\
Z_{k,2}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1\ et\ l'_2 = l_2 - 1\} ; \\
Z_{k,3}(\vec{l}) &= \{\vec{l'}\ : l'_1 = l_1,\ l'_2 = l_2\ \ et\ (l'_3,\ l'_4) \preceq (l_3,\ l_4)^t\ \}.
\end{aligned}$$

Preuve : Comme $\vec{l'}$ produit un élément qui est consommé par $\vec{l}$ alors $\vec{l'} \preceq \vec{l}$. D'après la définition de l'opérateur $\preceq$, nous avons trois cas possibles :

1. $l'_1 < l_1$. D'après la *proposition 18* (*page 85*), $l'_1 = l_1 - 1$. C'est-à-dire que :

$$\vec{l'} \in Z_{k,1}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1 - 1\}$$

2. $l'_1 = l_1$ et $l'_2 < l_2$. D'après la *proposition 18* (*page 85*), $l'_2 = l_2 - 1$. C'est-à-dire que :

$$\vec{l'} \in Z_{k,2}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1 \text{ et } \ l'_2 = l_2 - 1\}$$

3. $l'_1 = l_1$, $l'_2 = l_2$ et $(l'_3,\ l'_4) \preceq (l_3,\ l_4)^t$. C'est-à-dire que :

$$\vec{l'} \in Z_{k,3}(\vec{l}) = \{\vec{l'} \ : l'_1 = l_1,\ l'_2 = l_2 \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t\}$$

Proposition 20 *Soient $\vec{l}$ une itération déroulée dans le domaine $\omega_2(D'_{k+1})$ et $E_{k,1}(\vec{l})$, $E_{k,2}(\vec{l})$ et $E_{k,3}(\vec{l})$ les ensembles d'itérations qui sont définis comme suit :*

$$\begin{aligned}
E_{k+1,1}(\vec{l}) &= \{\ \vec{l'} \in \omega_2(D'_k) |\ \exists\, \vec{l}'' \in \omega_2(D'_{k+1}) \ : l'_1 = l_1 - 1,\ l''_1 = l_1 \\
&\quad \text{et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'}\ \}\ ; \\
E_{k+1,2}(\vec{l}) &= \{\ \vec{l'} \in \omega_2(D'_k) |\ \exists\, \vec{l}'' \in \omega_2(D'_{k+1}) \ : l'_1 = l_1,\ l'_2 = l_2 - 1,\ l''_1 = l_1,\ l''_2 = l_2 \\
&\quad \text{et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'}\ \}\ ; \\
E_{k+1,3}(\vec{l}) &= \{\ \vec{l'} \in \omega_2(D'_k) |\ \exists\, \vec{l}'' \in \omega_2(D'_{k+1}) \ : l'_1 = l_1,\ l'_2 = l_2,\ l''_1 = l_1,\ l''_2 = l_2, \\
&\quad (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t \preceq (l''_3,\ l''_4)^t \text{ et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'}\ \}.
\end{aligned}$$

Les cardinaux $|E_{k,1}(\vec{l})|$, $|E_{k,2}(\vec{l})|$ et $|E_{k,3}(\vec{l})|$ sont bornés respectivement par $N_2^k * d_1^{k,max}$, $T_1^e * d_2^{k,max}$ et $T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$, avec :

1. *$T_1^e = a_{1,1}$ est la taille des grandes tuiles sur l'axe i ;*
2. *$T_2^e = a_{2,4}$ est la taille des grandes tuiles sur l'axe j ;*
3. *$T_1^r = a_{1,2}$ est la taille des petites tuiles sur l'axe i ;*
4. *$d_1^{k,max}$ est la plus grande projection sur l'axe i de toutes les dépendances de l'instruction S_k vers l'instruction S_{k+1} du code en* figure 5.5 *(*page 82*) ;*
5. *$d_2^{k,max}$ est la plus grande projection sur l'axe j de toutes les dépendances de l'instruction S_k vers l'instruction S_{k+1} du code en* figure 5.5 *(*page 82*) ;*
6. *N_2^k est le nombre d'itérations de la deuxième boucle du nid k du code en* figure 5.1 *(*page 68*).*

Preuve : Soit $\vec{d}$ la dépendance entre $\vec{i'} = \mathcal{AP}^{-1}\vec{l'}$ et $\vec{i}'' = \mathcal{AP}^{-1}\vec{l}''$. Sachant que $\vec{i'} + \vec{d} = \vec{i}''$.

1. $|E_{k+1,1}(\vec{l})|$? Nous savons que :

$$\begin{aligned}
i'_1 &= a_{1,1} * l'_1 + a_{1,2} * l'_3 + l'_5 \\
i'_2 &= a_{2,4} * l'_2 + a_{2,5} * l'_4 + l'_6 \\
i''_1 &= a_{1,1} * l''_1 + a_{1,2} * l''_3 + l''_5 \\
i''_2 &= a_{2,4} * l''_2 + a_{2,5} * l''_4 + l''_6
\end{aligned}$$

Comme $\vec{i'} + \vec{d} = \vec{i}''$, alors :

$$\begin{aligned} a_{1,1} * l_1' + a_{1,2} * l_3' + l_5' + d_1 &= a_{1,1} * l_1'' + a_{1,2} * l_3'' + l_5'' \\ a_{2,4} * l_2' + a_{2,5} * l_4' + l_6' + d_2 &= a_{2,4} * l_2'' + a_{2,5} * l_4'' + l_6'' \end{aligned}$$

D'après la définition de l'ensemble $E_{k+1,1}(\vec{l})$:

$$l_1' = l_1'' - 1$$

Comme $l_1' = l_1'' - 1$ et $a_{1,1} * l_1' + a_{1,2} * l_3' + l_5' + d_1 = a_{1,1} * l_1'' + a_{1,2} * l_3'' + l_5''$ alors :

$$a_{1,2} * l_3' + l_5' = a_{1,2} * l_3'' + l_5'' + a_{1,1} - d_1$$

Nous savons que $0 \leq a_{1,2} * l_3'' + l_5'' \leq a_{1,1} - 1$. Par conséquent :

$$a_{1,1} - d_1 \leq a_{1,2} * l_3' + l_5' \leq 2 * a_{1,1} - d_1 - 1$$

Comme $a_{1,2} * l_3' + l_5' \leq a_{1,1} - 1$ et $0 \leq d_1 \leq a_{1,1}$ alors :

$$a_{1,1} - d_1 \leq a_{1,2} * l_3' + l_5' \leq a_{1,1} - 1$$

Comme nous cherchons à prendre toutes les itérations $\vec{l'}$ qui sont susceptibles d'appartenir à $E_{k+1,1}(\vec{l})$, la contrainte précédente peut être remplacée par :

$$a_{1,1} - d_1^{k,max} \leq a_{1,2} * l_3' + l_5' \leq a_{1,1} - 1$$

Par conséquent, les seules conditions que doit satisfaire $\vec{l'}$ sont :

(a) $l_1' = l_1 - 1$: C'est-à-dire que toutes les tuiles qui ont la coordonnée $l_1 - 1$ sur l'axe i sont concernées. Le nombre total de ces tuiles est $N_2^k / a_{2,4}$.

(b) $a_{1,1} - d_1^{k,max} \leq a_{1,2} * l_3' + l_5' \leq a_{1,1} - 1$: C'est-à-dire que dans la même tuile, les itérations qui sont concernées sont celles qui satisfont cette contrainte. Le nombre total de ces itérations est $a_{2,4} * d_1^{k,max}$.

Par conséquent, le nombre total d'itérations $\vec{l'} \in E_{k+1,1}(\vec{l})$ est borné par :

$$(N_2^k / a_{2,4}) * (a_{2,4} * d_1^{k,max}) = N_2^k * d_1^{k,max}$$

2. $|E_{k+1,2}(\vec{l})|$? De la même manière que la cas précédent, nous obtenons :

$$\begin{aligned} a_{1,1} * l_1' + a_{1,2} * l_3' + l_5' + d_1 &= a_{1,1} * l_1'' + a_{1,2} * l_3'' + l_5'' \\ a_{2,4} * l_2' + a_{2,5} * l_4' + l_6' + d_2 &= a_{2,4} * l_2'' + a_{2,5} * l_4'' + l_6'' \end{aligned}$$

D'après la définition de l'ensemble $E_{k+1,2}(\vec{l})$:

$$l_1' = l_1'' \text{ et } l_2' = l_2'' - 1$$

Par conséquent :

$$\begin{aligned} a_{1,2} * l_3' + l_5' &= a_{1,2} * l_3'' + l_5'' - d_1 \\ a_{2,5} * l_4' + l_6' &= a_{2,5} * l_4'' + l_6'' + a_{2,4} - d_2 \end{aligned}$$

Nous savons que $0 \leq a_{1,2} * l_3^{''} + l_5^{''} \leq a_{1,1} - 1$ et $0 \leq a_{2,5} * l_4^{''} + l_6^{''} \leq a_{2,4} - 1$, donc :

$$\begin{array}{rcl} -d_1 & \leq & a_{1,2} * l'_3 + l'_5 \leq a_{1,1} - d_1 - 1 \\ a_{2,4} - d_2 & \leq & a_{2,5} * l'_4 + l'_6 \leq 2 * a_{2,4} - d_2 - 1 \end{array}$$

Comme $a_{1,2} * l_3^{''} + l_5^{''} \geq 0$, $0 \leq d_2 \leq a_{2,4}$, $0 \leq d_1 \leq a_{1,1}$ et $a_{2,5} * l'_4 + l'_6 \leq a_{2,4}$, alors :

$$\begin{array}{rcl} 0 & \leq & a_{1,2} * l'_3 + l'_5 \leq a_{1,1} - 1 \\ a_{2,4} - d_2 & \leq & a_{2,5} * l'_4 + l'_6 \leq a_{2,4} - 1 \end{array}$$

Comme nous cherchons à prendre toutes les itérations qui sont susceptibles d'appartenir à $E_{k+1,2}(\vec{l})$, les deux contraintes précédentes peuvent être remplacées par :

$$\begin{array}{rcl} 0 & \leq & a_{1,2} * l'_3 + l'_5 \leq a_{1,1} - 1 \\ a_{2,4} - d_2^{k,max} & \leq & a_{2,5} * l'_4 + l'_6 \leq a_{2,4} - 1 \end{array}$$

Par conséquent, les seules conditions que doit satisfaire $\vec{l'}$ sont :

(a) $l'_1 = l_1$ et $l'_2 = l_2 - 1$: C'est-à-dire que les itérations $\vec{l'}$ qui sont concernées, sont situées dans la même tuile.

(b) $0 \leq a_{1,2} * l'_3 + l'_5 \leq a_{1,1} - 1$ et $a_{2,4} - d_2^{k,max} \leq a_{2,5} * l'_4 + l'_6 \leq a_{2,4} - 1$: C'est-à-dire que dans cette même tuile, les itérations $\vec{l'}$ qui sont concernées, sont celles qui satisfont les deux contraintes précédentes. Le nombre total de ces itérations est donné par $a_{1,1} * d_2^{k,max}$.

Par conséquent, le nombre total d'itérations $\vec{l'} \in E_{k+1,2}(\vec{l})$ est borné par :

$$a_{1,1} * d_2^{k,max} = T_1^e * d_2^{k,max}$$

3. $|E_{k+1,3}(\vec{l})|$? D'après la définition de l'ensemble $E_{k+1,3}(\vec{l})$, $|E_{k+1,3}(\vec{l})|$ représente le nombre de données vivantes qui sont produites dans la même grande tuile. Soit $(l_3,\ l_4)^t$ les coordonnées d'une petite tuile b dans une grande tuile a. La plus ancienne petite tuile dans a qui peut produire une donnée qui sera utilisée dans b est une tuile c de coordonnées $(l'_3,\ l'_4)^t$ avec :

$$l'_3 = l_3 - \lceil \frac{d_1^{k,max}}{T_1^r} \rceil$$

De la même manière que la fusion, nous pouvons vérifier que toutes les données vivantes sont produites entre $(l'_3,\ l'_4)^t$ et $(l_3,\ l_4)^t$. Le nombre d'itérations entre ces deux itérations est borné par :

$$\frac{T_2^e}{T_2^r} * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$$

Comme chaque itération $(l'_3,\ l'_4)^t$ produit $T_1^r * T_2^r$ données, le nombre de données vivantes qui sont produites entre $(l'_3,\ l'_4)^t$ et $(l_3,\ l_4)^t$ est borné par :

$$T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$$

Par conséquent, le nombre total d'itérations $\vec{l'} \in E_{k+1,3}(\vec{l})$ est borné par :

$$T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$$

□

5.3.4 Allocation des tampons dans le cas à deux dimensions

Théorème 13 *$\forall \vec{l} \in \omega_2(D'_{k+1})$, le tableau A_k peut être remplacé par 5 tampons :*

1. *un tampon $B_{k,1}$ de taille $N_2^k * d_1^{k,max}$ qui contiendra les données produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que :*

$$l'_1 = l_1 - 1 \text{ et } l_1^{''} = l_1$$

2. *un tampon $B'_{k,1}$ de taille $N_2^k * d_1^{k,max}$ qui contiendra les données produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que :*

$$l'_1 = l_1 \text{ et } l_1^{''} = l_1 + 1$$

3. *un tampon $B_{k,2}$ de taille $T_1^e * d_2^{k,max}$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que :*

$$l'_1 = l_1,\ l'_2 = l_2 - 1,\ l_1^{''} = l_1 \text{ et } l_2^{''} = l_2$$

4. *un tampon $B'_{k,2}$ de taille $T_1^e * d_2^{k,max}$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que*

$$l'_1 = l_1,\ l'_2 = l_2,\ l_1^{''} = l_1 \text{ et } l_2^{''} = l_2 + 1$$

5. *un tampon $B_{k,3}$ de taille $T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$ qui contiendra les données du tableau A_k produites par des $\vec{l'}$ et qui seront consommées par des $\vec{l}''$ telles que :*

$$l'_1 = l_1,\ l'_2 = l_2,\ l_1^{''} = l_1,\ l_2^{''} = l_2 \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t \preceq (l_3^{''},\ l_4^{''})^t$$

Preuve : Nous devons montrer que n'importe quelle donnée vivante du tableau A_k à n'importe quelle itération déroulée $\vec{l}$, doit se trouver dans les tampons $B_{k,1}$, $B'_{k,1}$, $B_{k,2}$, $B'_{k,2}$ ou $B_{k,3}$. Soit $E_{k+1}(\vec{l})$ l'ensemble des données vivantes du tableau A_k à une itération déroulée $\vec{l}$:

$$\begin{aligned} E_{k+1}(\vec{l}) \quad = \quad & \{\vec{l'} \in \omega_2(D'_k) |\ \exists\ \vec{l}'' \in \omega_2(D'_{k+1}) \ : \\ & \vec{l'} \preceq \vec{l} \preceq \vec{l}'' \text{ et } \vec{l}'' \text{ consomme la donnée produite par } \vec{l'}\ \} \end{aligned}$$

D'après la définition de l'opérateur $\preceq$, la relation $\vec{l'} \preceq \vec{l}$ donne trois cas :

1. $l'_1 < l_1$. Comme $\vec{l'} \preceq \vec{l} \preceq \vec{l}''$ alors :

$$l'_1 \le l_1 \le l_1^{''} \Rightarrow l'_1 < l_1 \le l_1^{''}$$

Comme $\vec{l}''$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 18* (*page 85*) :

$$l'_1 \ge l_1^{''} - 1$$

Comme $l'_1 \ge l_1^{''} - 1$ et $l'_1 < l_1 \le l_1^{''}$, alors :

$$l_1^{''} - 1 \le l'_1 < l_1 \le l_1^{''}$$

Cela signifie que :

$$l'_1 = l_1 - 1 \text{ et } l_1 = l_1^{''}$$

C'est-à-dire que :

$$\vec{l'} \in E_{k+1,1}(\vec{l}) = \{\vec{l'} \in \omega_2(D'_k) | \exists \vec{l}^{''} \in \omega_2(D'_{k+1}) : l'_1 = l_1 - 1, \ l_1^{''} = l_1 \text{ et } \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \}$$

D'après la *proposition 20* (*page 86*), $|E_{k+1,1}(\vec{l})|$ est borné par $N_2^k * d_1^{k,max}$. Comme chaque itération déroulée produit un seul élément du tableau A_k, alors toutes les données du tableau A_k qui sont produites par des $\vec{l'} \in E_{k+1,1}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,1}$ de taille $N_2^k * d_1^{k,max}$.

2. $l'_1 = l_1$ et $l'_2 < l_2$. Comme $\vec{l'} \preceq \vec{l}^{''}$, alors $l'_1 \leq l_1^{''}$.
Comme $\vec{l}^{''}$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 18* (*page 85*) :

$$l'_1 \geq l_1^{''} - 1$$

Par conséquent :

$$l'_1 \leq l_1^{''} \leq l'_1 + 1$$

Donc, nous avons deux sous-cas :

(a) $l_1^{''} = l'_1$.
Comme $l_1^{''} = l'_1 = l_1$ et $\vec{l} \preceq \vec{l}^{''}$ alors :

$$l_2^{''} \geq l_2$$

Comme $\vec{l}^{''}$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 18* (*page 85*) :

$$l_2^{''} \leq l'_2 + 1$$

Comme $l'_2 < l_2$, $l_2^{''} \geq l_2$ et $l_2^{''} \leq l'_2 + 1$ alors :

$$l'_2 < l_2 \leq l_2^{''} \leq l'_2 + 1$$

Cela signifie que

$$l'_2 = l_2 - 1 \text{ et } l_2{}^{''} = l_2$$

C'est-à-dire que :

$$\vec{l'} \in E_{k+1,2}(\vec{l}) = \{\vec{l'} \in \omega_2(D'_k) | \exists \vec{l}^{''} \in \omega_2(D'_{k+1}) : l'_1 = l_1, \ l'_2 = l_2 - 1, \ l_1^{''} = l_1, \ l_2^{''} = l_2 \text{ et } \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \}$$

D'après la *proposition 20* (*page 86*), $|E_{k+1,2}(\vec{l})|$ est borné par $T_1^e * d_2^{k,max}$. Par conséquent, toutes les données qui sont produites par des $\vec{l'} \in E_{k+1,2}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,2}$ de taille $T_1^e * d_2^{k,max}$.

(b) $l_1'' = l_1' + 1$. C'est-à-dire que :

$$\begin{aligned}\vec{l'} \in E'_{k+1,1}(\vec{l}) \;=\; & \{\vec{l'} \in \omega_2(D'_k) |\; \exists\, \vec{l}'' \in \omega_2(D'_{k+1}) \;: l_1' = l_1,\; l_1'' = l_1 + 1 \text{ et} \\ & \vec{l}'' \text{ consomme la donnée produite par } \vec{l'} \;\}\end{aligned}$$

En posant $\vec{m} = \vec{l} + (1,\ 0,\ 0,\ 0)^t$, nous obtenons $E'_{k+1,1}(\vec{l}) = E_{k+1,1}(\vec{m})$. D'après la *proposition 20* (*page 86*),$|E_{k+1,1}(\vec{m})|$ est borné par $N_2^k * d_1^{k,max}$. Par conséquent, toutes les données du tableau A_k qui sont produites par des $\vec{l'} \in E'_{k+1,1}(\vec{l})$ peuvent être stockées dans un tampon $B'_{k,1}$ de taille $N_2^k * d_1^{k,max}$.

3. $l_1' = l_1$ et $l_2' = l_2$. Comme $\vec{l'} \preceq \vec{l}''$ alors $l_1' \le l_1''$.
Comme $\vec{l}''$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 18* (*page 85*) :

$$l_1' \ge l_1'' - 1$$

Par conséquent :

$$l_1' \le l_1'' \le l_1' + 1$$

Donc nous avons deux sous-cas :

(a) $l_1'' = l_1'$.
Comme $l_1 = l_1''$ et $\vec{l} \preceq \vec{l}''$ alors $l_2'' \ge l_2$. Donc nous avons deux cas :

i. $l_2'' = l_2$. Comme $\vec{l'} \preceq \vec{l} \preceq \vec{l}''$, $l_1' = l_1 = l_1''$ et $l_2' = l_2 = l_2''$, alors :

$$(l_3',\ l_4')^t \preceq (l_3,\ l_4)^t \preceq (l_3'',\ l_4'')^t$$

C'est-à-dire que :

$$\begin{aligned}\vec{l'} \in E_{k+1,3}(\vec{l}) \;=\; & \{\vec{l'} \in \omega_2(D'_k) |\; \exists\, \vec{l}'' \in \omega_2(D'_{k+1}) \;: l_1' = l_1 = l_1'',\; l_2' = l_2 = l_2'', \\ & (l_3',\ l_4')^t \preceq (l_3,\ l_4)^t \preceq (l_3'',\ l_4'')^t \text{ et } \vec{l}'' \text{ consomme} \\ & \text{la donnée produite par } \vec{l'} \;\}\end{aligned}$$

D'après la *proposition 20* (*page 86*), $|E_{k+1,3}(\vec{l})|$ est borné par $T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$. Par conséquent, toutes les données du tableau A_k qui sont produites par des $\vec{l'} \in E_{k+1,3}(\vec{l})$ peuvent être stockées dans un tampon $B_{k,3}$ de taille $T_1^r * T_2^e * (\lceil \frac{d_1^{k,max}}{T_1^r} \rceil + 1)$.

ii. $l_2'' > l_2$.
Comme $\vec{l}''$ consomme l'élément produit par $\vec{l'}$, alors d'après la *proposition 18* (*page 85*) :

$$l_2'' \le l_2' + 1$$

Par conséquent :

$$l_2 < l_2'' \le l_2' + 1 = l_2 + 1$$

Cela signifie que :

$$l_2^{''} = l_2 + 1 \text{ et } l_2' = l_2$$

C'est-à-dire que :

$$\begin{aligned}\vec{l'} \in E'_{k+1,2}(\vec{l}) \;\; = \;\; & \{\vec{l'} \in \omega_2(D'_k) |\ \exists\, \vec{l}^{''} \in \omega_2(D'_{k+1}) \ : l_1' = l_1 = l_1^{''},\ l_2' = l_2, \\ & l_2^{''} = l_2 + 1 \ \text{ et } \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \ \}\end{aligned}$$

En posant $\vec{m} = \vec{l} + (0,\ 1,\ 0,\ 0)^t$, nous obtenons $E'_{k+1,2}(\vec{l}) = E_{k+1,2}(\vec{m})$. D'après la *proposition 20* (*page 86*), $|E_{k+1,2}(\vec{m})|$ est borné par $T_1^e * d_2^{k,max}$. Par conséquent, toutes les données du tableau A_k qui sont produites par des $\vec{l'} \in E'_{k+1,2}(\vec{l})$ peuvent être stockées dans un tampon $B'_{k,2}$ de taille $T_1^e * d_2^{k,max}$.

(b) $l_1{}'' = l_1' + 1$. C'est-à-dire que :

$$\begin{aligned}\vec{l'} \in E'_{k+1,1}(\vec{l}) \;\; = \;\; & \{\vec{l'} \in \omega_2(D'_k) |\ \exists\, \vec{l}^{''} \in \omega_2(D'_{k+1}) \ : l_1' = l_1,\ l_1^{''} = l_1 + 1 \text{ et} \\ & \vec{l}^{''} \text{ consomme la donnée produite par } \vec{l'} \ \}\end{aligned}$$

En posant $\vec{m} = \vec{l} + (1,\ 0,\ 0,\ 0)^t$, nous obtenons $E'_{k,1}(\vec{l}) = E_{k,1}(\vec{m})$. D'après la *proposition 20* (*page 86*), $|E_{k+1,1}(\vec{m})|$ est borné par $N_2^k * d_1^{k,max}$. Par conséquent, toutes les données du tableau A_k qui sont produites par des $\vec{l'} \in E'_{k+1,1}(\vec{l})$ peuvent être stockées dans un tampon $B'_{k,1}$ de taille $N_2^k * d_1^{k,max}$.

5.3.5 Allocation des registres dans le cas à deux dimensions

Comme nous l'avons signalé précédemment, les petites tuiles sont utilisées pour l'allocation des registres. Après le déroulage des boucles l_5 et l_6 dans le code en *figure 5.5* (*page 82*), boucles qui parcourent les itérations dans les petites tuiles, nous obtenons le code en *figure 5.7* (*page 84*). À chaque nouvelle itération $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ dans ce code, nous produisons $T_1^r * T_2^r$ éléments du tableau A_k. Chacun de ces éléments sera sauvegardé dans un registre. Par conséquent, nous avons besoin de $T_1^r * T_2^r$ registres. Certains de ces registres seront vivants aux prochaines itérations $\vec{l'} = (l_1',\ l_2',\ l_3',\ l_4')^t$ qui suivent l'itération $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ telles que :

1. $l_1' = l_1$;
2. $l_2' = l_2$;
3. $l_3' = l_3$;
4. $l_4' > l_4$

C'est-à-dire que à l'itération $\vec{l'} = (l_1',\ l_2',\ l_3',\ l_4')^t$, un nouvel élément du tableau A_{k+1} sera calculé en fonction d'un patron de données du tableau A_k et l'une de ces données est produite par l'itération $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$.

Afin d'exploiter ces registres vivants dans les prochaines itérations qui suivent $\vec{l}$, nous devons allouer $T_1^r * T_2^r + Nbr$ registres :

1. $T_1^r * T_2^r$ est le nombre de données du tableau A_k que nous produisons à chaque nouvelle itération $\vec{l}$;
2. Nbr est le nombre de registre vivants.

Définition 31 (Registre vivant) *Soit* $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ *une itération du code en* figure 5.7 (page 84)*. Un registre qui contient une donnée du tableau* A_k *est dit vivant à l'itération* $\vec{l}$ *si et seulement si :*

- *la donnée du tableau* A_k *qu'il contient est produite par une itération déroulée* $\vec{l'}$ *telle que :* $l'_1 = l_1,\ l'_2 = l_2,\ l'_3 = l_3$ *et* $l'_4 < l_4$
- *il existe une itération déroulée* $\vec{l}''$ *qui va le consommer telle que* $l_1'' = l_1,\ l_2'' = l_2,\ l_3'' = l_3$ *et* $l_4'' \geq l_4$.

Comme nous pouvons le remarquer, les registres vivants sont pris en considération uniquement dans la même série de tuiles internes qui est située dans la même tuile externe.

Proposition 21 *Soit* $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ *une itération du code en* figure 5.7 (page 84)*. Le nombre de registres qui contiennent des données du tableau* A_k *et qui sont vivants à* $\vec{l}$ *est borné par :*

$$d_2^{k,\ max,\ c} * T_2^r$$

Avec $d_2^{k,\ max,\ c}$ *est la plus grande projection de toutes les dépendances sur l'axe* j *qui ont une projection sur l'axe* i *inférieure à la taille des petites tuiles sur cet axe* (T_1^r)*.*

Preuve : La preuve se fait de la même manière que la *proposition 20* (*page 86*).

5.3.6 Génération du code dans le cas à deux dimensions

Le tableau A_k sera remplacé par les tampons $B_{k,1}$, $B'_{k,1}$, $B_{k,2}$, $B'_{k,2}$ et $B_{k,3}$ et $T_1^r * T_2^r + d_2^{k,\ \mathrm{max},\ c} * T_2^r$ registres. À n'importe quelle itération $\vec{l}$, nous devons savoir dans quel tampon ou dans quel registre la donnée du tableau A_k sera rangée ou récupérée.

1) Gestion des tampons : Soit $\vec{l} \in \omega_2(D'_{k+1})$ une itération déroulée dans le code en *figure 5.7* (*page 84*). D'après la *proposition 19* (*page 85*), une itération déroulée $\vec{l'} \in \omega_2(D'_k)$ qui produit une donnée du tableau A_k qui sera consommée par l'itération déroulée $\vec{l}$ peut se trouver dans 3 zones possibles :

1. $Z_{k,1}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1 - 1\}$;
2. $Z_{k,2}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1 \text{ et } l'_2 = l_2 - 1\}$;
3. $Z_{k,3}(\vec{l}) = \{\vec{l'}\ : l'_1 = l_1,\ l'_2 = l_2 \text{ et } (l'_3,\ l'_4)^t \preceq (l_3,\ l_4)^t\}$.

D'après le *théorème 13* (*page 89*), les données qui sont produites dans :

1. $Z_{k,1}(\vec{l})$ se trouvent dans le tampon $B_{k,1}$;
2. $Z_{k,2}(\vec{l})$ se trouvent dans le tampon $B_{k,2}$;
3. $Z_{k,3}(\vec{l})$ se trouvent dans le tampon $B_{k,3}$.

Mais chaque nouvelle donnée du tableau A_k produite à une itération déroulée $\vec{l'}$, sera sauvegardée dans l'un (ou plusieurs) des tampons suivants : $B'_{k,1}$, $B'_{k,2}$, $B_{k,3}$ (*théorème 13* en (*page 89*)).

Donc, à une itération déroulée $\vec{l}$, pour utiliser une donnée du tableau A_k produite par une itération déroulée $\vec{l'}$, nous devons la récupérer dans les tampons $B_{k,1}$, $B_{k,2}$ ou $B_{k,3}$. Mais l'itération $\vec{l'}$ sauvegarde cette donnée dans les tampons $B'_{k,1}$, $B'_{k,2}$ ou $B_{k,3}$ (ou plusieurs). Par conséquent nous devons :

1. transférer tous les éléments du tampon $B'_{k,2}$ dans le tampon $B_{k,2}$ lorsque nous passons d'une grande tuile vers la prochaine. Car la donnée qui est sauvegardée dans le tampon $B'_{k,2}$ à une itération déroulée $\vec{l'}$, sera récupérée par une itération déroulée $\vec{l}$ ($l_1 = l'_1$ et $l_2 = l'_2 + 1$) dans le tampon $B_{k,2}$;
2. transférer tous les éléments du tampon $B'_{k,1}$ dans le tampon $B_{k,1}$ lorsque nous passons d'une série de grandes tuiles vers la prochaine. Car la donnée qui est sauvegardée dans le tampon $B'_{k,1}$ à une itération déroulée $\vec{l'}$, sera récupérée par une itération $\vec{l}$ ($l_1 = l'_1+1$) dans le tampon $B_{k,1}$.

Cette méthode de génération du code n'est pas efficace, à cause des copies des tampons $B'_{k,1}$ dans le tampon $B_{k,1}$ et $B'_{k,2}$ dans le tampon $B_{k,2}$. De la même manière que le chapitre précédent, pour palier ce problème, il suffit de fusionner les tampons :

1. $B_{k,1}$ et $B'_{k,1}$ en un seul tampon qui sera noté par $BT_{k,1}[2]$ telles que $|BT_{k,1}[0]| = |BT_{k,1}[1]| = |B_{k,1}| = |B'_{k,1}|$. Dans ce cas, une itération déroulée $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,1}$ va la stocker dans $BT_{k,1}[l_1 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,1}$, elle va la récupérer dans $BT_{k,1}[(l_1 - 1) \bmod 2]$.
2. $B_{k,2}$ et $B'_{k,2}$ en un seul tampon qui sera noté par $BT_{k,2}[2]$ telles que $|BT_{k,2}[0]| = |BT_{k,2}[1]| = |B_{k,2}| = |B'_{k,2}|$. Dans ce cas une itération déroulée $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,2}$ va la stocker dans $BT_{k,2}[l_2 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,2}$, elle va la récupérer dans $BT_{k,2}[(l_2 - 1) \bmod 2]$.

2) Gestion des registre : Soit $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4)^t$ une itération dans le code *figure 5.7* (*page 84*). Les $T_1^r * T_2^r$ éléments du tableau A_k produits à cette itération seront sauvegardés dans $T_1^r * T_2^r$ registres différents (sachant qu'ils sont sauvegardés dans un ou plusieurs tampons définis précédemment). Comme nous l'avons mentionné précédemment, en plus de ces registres, nous avons besoin de $d_2^{k,max,c} * T_1^r$ registres supplémentaires qui contiendront les données du tableau A_k qui sont produites par des itérations $\vec{l'} \prec \vec{l}$ ($l'_1 = l_1$, $l'_2 = l_2$, $l'_3 = l_3$ et $l'_4 < l_4$) et qui sont susceptibles d'être utilisées par des itérations $\vec{l}'' \succeq \vec{l}$ ($l_1'' = l_1$, $l_2'' = l_2$, $l_3'' = l_3$ et $l_4'' \geq l_4$). Ces données sont produites dans $m = \lceil \frac{d_2^{k,max,c}}{T_2^r} \rceil$ itérations differentes (le nombre d'itérations $\vec{l'}$ possibles). Ces différentes itérations seront notées par :

$$\vec{l}^1, \cdots, \vec{l}^m \ \text{ avec } \vec{l}^1 \prec \cdots \prec \vec{l}^m$$

Nous pouvons vérifier facilement que :

1. $\forall i \in [2 \cdots m]\ :\ \vec{l}^i$ produit $T_1^r * T_2^r$ éléments du tableau A_k ;
2. $\vec{l}^1$ produit $\begin{cases} T_1^r * T_2^r \text{ éléments} & \text{si} \quad (d_2^{k,max,c} \bmod T_2^r) = 0 \\ (d_2^{k,max,c} \bmod T_2^r) * T_1^r \text{ éléments} & \text{sinon} \end{cases}$

Nous notons l'itération $\vec{l}$ par $\vec{l}^{m+1}$ et l'itération qui la suit par $\vec{l}^{m+2}$, c'est-à-dire que $\vec{l}^{m+2} = \vec{l} + (0,\ 0,\ 0,\ 1)^t$. Lorsque nous passons de l'itération $\vec{l}$ vers $\vec{l}^{m+2}$, nous devons mettre à jour les différents registres de la manière suivante :

si $(d_2^{k,max,c} \bmod T_2^r) = 0$ alors
  les registres associés à $\vec{l^1}$ vont recevoir les registres associés à $\vec{l^2}$
sinon
  ils vont recevoir une partie des registres associés à $\vec{l^2}$
finsi
pour $i = 2$ à $i = m$
  les registres associés à $\vec{l^i}$ vont recevoir une partie des registre associés à $\vec{l^{i+1}}$
finpour

3) Écriture et lecture des éléments des différents tableaux A_k : Comme nous l'avons mentionné précédemment, à chaque itération déroulée $\vec{l}$, nous calculons un nouvel élément du tableau A_{k+1} en fonction d'un patron de données du tableau A_k. Ce nouvel élément du tableau A_{k+1} sera toujours rangé dans le tampon $B_{k,3}$ et dans l'un des $T_1^r * T_2^r$ registres associés à $\vec{l}$. Par contre, il sera rangé dans les tampons :

1. $B'_{k,1}$ si $T_1^r * l_3 + l_5 \geq T_1^e - d_1^{k,max}$;
2. $B'_{k,2}$ si $T_2^r * l_4 + l_6 \geq T_2^e - d_2^{k,max}$.

Un élément du tableau A_k qui est nécessaire pour le calcul de l'élément du tableau A_{k+1} à l'itération $\vec{l}$ sera récupéré dans l'un des registres qui sont associés ou vivant à l'itération $\vec{l}$ ou dans l'un des tampons $B_{k,1}$, $B_{k,2}$ ou $B_{k,3}$. Soit $\vec{l'}$ l'itération déroulée qui produit cet élément du tableau A_k :

1. si $l'_1 = l_1 - 1$ alors il sera récupéré dans le tampon $B_{k,1}$;
2. si $l'_1 = l_1$ et $l'_2 = l_2 - 1$ alors il sera récupéré dans le tampon $B_{k,2}$;
3. si $l'_1 = l_1$, $l'_2 = l_2$ et $l'_3 < l_3$ alors il sera récupéré dans le tampon $B_{k,3}$;
4. si $l'_1 = l_1$, $l'_2 = l_2$, $l'_3 = l_3$ et $l'_4 < l_4$ alors il sera récupéré dans l'un des registres qui sont vivants à l'itération $\vec{l}$;
5. si $l'_1 = l_1$, $l'_2 = l_2$, $l'_3 = l_3$ et $l'_4 = l_4$ alors il sera récupéré dans l'un des registres qui sont associés à l'itération $\vec{l}$.

5.3.7 Cas général

Ici, nous traitons le cas général, c'est-à-dire nous supposons que la profondeur des nids du code en entrée et la matrice de permutation $\mathcal{P}$ sont quelconques. De la même manière que le cas à deux dimensions, chaque tableau A_k sera remplacé par plusieurs tampons et registres.

Définition 32 *Le vecteur d'itération $\vec{l}$ du code en* figure 5.4 (page 72) *peut être décomposé en trois sous vecteurs :*

1. $\vec{l^{\mathcal{E}}} = \mathcal{E}\ \vec{i_{\mathcal{E}}}$ *qui parcourt les grandes tuiles ;*
2. $\vec{l^{\mathcal{R}}} = \mathcal{R}\ \vec{i_{\mathcal{R}}}$ *qui parcourt le petites tuiles ;*
3. $\vec{l^{\mathcal{S}}} = \mathcal{S}\ \vec{i_{\mathcal{S}}}$ *qui parcourt les itérations des petites tuiles.*

Déroulage des boucles les plus internes

Comme nous l'avons mentionné précédemment, la première étape qui précède l'allocation des registres et des tampons est le déroulage [15, 66, 16] entier des boucles $\vec{l^{\mathcal{S}}}$ qui parcourent les itérations des petites tuiles. Cela revient à :

1. supprimer les boucles $\vec{l^{\mathcal{S}}}$;
2. dupliquer les instructions S_1, S_2,$\cdots$ et S_p , autant de fois que le nombre de valeurs que peut prendre le vecteur d'itération $\vec{l^{\mathcal{S}}}$.

Comme $\vec{0} \leq \vec{l^{\mathcal{S}}} \leq \vec{T^r} - \vec{1}$, alors le code en *figure 5.4* (*page 72*) sera transformé en code de la *figure 5.8* (*page 96*).

do($\vec{l^{\mathcal{E}}}$)
  do($\vec{l^{\mathcal{R}}}$)
    $\vec{l} = (\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}},\ \vec{0})^t$
    $C_1(\vec{l}):\ A_1(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$
    .
    $C_k(\vec{l}):\ A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)$
    .
    $C_p(\vec{l}):\ A_p(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)$
    .
    .
    $\vec{l} = (\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}},\ \vec{m})^t$
    $C_1(\vec{l}):\ A_1(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$
    .
    $C_k(\vec{l}):\ A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)$
    .
    $C_p(\vec{l}):\ A_n(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)$
    .
    .
    $\vec{l} = (\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}},\ \vec{T^r} - \vec{1})^t$
    $C_1(\vec{l}):\ A_1(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1) = A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_1^1) \otimes \cdots \otimes A_0(\mathcal{AP}^{-1}\vec{l} - \vec{h}_1 + \vec{v}_{m_1}^1)$
    .
    $C_k(\vec{l}):\ A_k(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k) = A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_1^k) \otimes \cdots \otimes A_{k-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_k + \vec{v}_{m_k}^k)$
    .
    $C_p(\vec{l}):\ A_p(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p) = A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_1^p) \otimes \cdots \otimes A_{p-1}(\mathcal{AP}^{-1}\vec{l} - \vec{h}_p + \vec{v}_{m_p}^p)$
  enddo
enddo

Figure 5.8: Code après le déroulage des boucles $\vec{l^{\mathcal{S}}}$

Définition 33 *Soit* $(\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}})^t$ *une itération du code* figure 5.8 (page 96) *et* $\vec{l^{\mathcal{S}}} = \vec{a}$. *Dans la suite de ce chapitre, nous appelons :*

1. $(\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}})^t$ *une itération* $\vec{l}$ *du code en* figure 5.8 ;
2. $(\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}},\ \vec{l^{\mathcal{S}}})^t$ *une itération déroulée* $\vec{l}$ *du code en* figure 5.8.

5.3.8 Allocation des tampons dans le cas général

De la même manière que le cas à deux dimensions le tableau A_k sera remplacé par $2n+1$ tampons :

- un tampon $B_{k,1}$ de taille $\mathcal{E}_1\ \vec{d}^{k,max} * \prod_{i=2}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1 - 1$ et $l''_1 = l_1$;
- un tampon $B'_{k,1}$ de taille $\mathcal{E}_1\ \vec{d}^{k,max} * \prod_{i=2}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1$ et $l''_1 = l_1 + 1$;

$$\vdots$$

$$\vdots$$

- un tampon $B_{k,m}$ de taille $\prod_{i=1}^{m-1}(\mathcal{E}_i\ \vec{T}^e) * (\mathcal{E}_m\ \vec{d}^{k,max}) * \prod_{i=m+1}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1, \cdots, l'_{m-1} = l_{m-1}$, $l'_m = l_m - 1$, $l''_1 = l_1, \cdots$ et $l''_m = l_m$;
- un tampon $B'_{k,m}$ de taille $\prod_{i=1}^{m-1}(\mathcal{E}_i\ \vec{T}^e) * (\mathcal{E}_m\ \vec{d}^{k,max}) * \prod_{i=m+1}^{n}(\mathcal{E}_i\ \vec{N}^k)$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1, \cdots, l'_m = l_m$, $l''_1 = l_1, \cdots, l''_{m-1} = l_{m-1}$ et $l''_m = l_m + 1$;

$$\vdots$$

$$\vdots$$

- un tampon $B_{k,n}$ de taille $\prod_{i=1}^{n-1}(\mathcal{E}_i\ \vec{T}^e) * (\mathcal{E}_n\ \vec{d}^{k,max})$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1, \cdots, l'_{n-1} = l_{n-1}$, $l'_n = l_n - 1$, $l''_1 = l_1, \cdots, l''_n = l_n$;
- un tampon $B'_{k,n}$ de taille $\prod_{i=1}^{n-1}(\mathcal{E}_i\ \vec{T}^e) * (\mathcal{E}_n\ \vec{d}^{k,max})$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1, \cdots, l'_n = l_n$, $l''_1 = l_1, \cdots, l''_{n-1} = l_{n-1}$ et $l''_n = l_n + 1$;
- un tampon $B_{k,n+1}$ de taille $(\lceil \frac{\mathcal{R}_1 d^{k,max}}{\mathcal{R}_1 T^r} \rceil + 1) * \prod_{k=2}^{n}(\mathcal{R}_k\ \vec{T}^e)$ qui contiendra les données du tableau A_k qui sont produites par des $\vec{l}'$ et qui seront consommées par des $\vec{l}''$ telles que $l'_1 = l_1, \cdots, l'_n = l_n$, $l''_1 = l_1, \cdots, l''_n = l_n$ et $(l'_{n+1}, \cdots,\ l'_{2n})^t \preceq (l_{n+1}, \cdots,\ l_{2n})^t \preceq (l''_{n+1}, \cdots,\ l''_{2n})^t$.

Avec :

1. $\vec{d}^{k,max}$ est le vecteur des plus grandes projections de toutes les dépendances de l'instruction S_k vers l'instruction S_{k+1} du code en *figure 5.4* (*page 72*) ;
2. $\vec{T}^e$ est le vecteur des tailles des tuiles externes ;
3. $\vec{T}^r$ est le vecteur des tailles des tuiles internes ;
4. $\vec{N}^k$ est le vecteur du nombre d'itérations de chaque boucle du *nid k* du code en *figure 5.1* (*page 68*) ;
5. $\mathcal{E}_i$ est la $i^{\text{ème}}$ ligne de la matrice $\mathcal{E}$ de permutation des boucles externes ;
6. $\mathcal{R}_i$ est la $i^{\text{ème}}$ ligne de la matrice $\mathcal{R}$ de permutation des boucles internes.

5.3.9 Allocation des registres dans le cas général

De la même manière que le cas à deux dimensions, les tuiles les plus internes sont réservées pour l'allocation des registres. Après le déroulage des boucles $\vec{l}^S$ dans le code en *figure 5.4* (*page 72*) (les boucles qui parcourent les itérations dans les tuiles internes), nous obtenons le code en *figure 5.8* (*page 96*). À chaque nouvelle itération $\vec{l} = (\vec{l}^{\mathcal{E}},\ \vec{l}^{\mathcal{R}})^t = (l_1, \cdots,\ l_{2n})$ dans ce code, nous produisons $\prod_{i=1}^{n} T_i^r$ nouveaux éléments du tableau A_k. Chacun de ces éléments sera sauvegardé dans un registre. Par conséquent, nous avons besoin de $\prod_{i=1}^{n} T_i^r$ registres. Certains de ces registres sont vivants aux prochaines itérations $\vec{l'} = (l'_1, \cdots, l'_{2n})^t$ qui suivent l'itération $\vec{l}$, telles que :

- $l_1 = l'_1$
- .
- .
- $l_{2n-1} = l'_{2n-1}$
- $l_{2n} < l'_{2n}$

C'est-à-dire qu'à l'itération $\vec{l'}$, un nouvel élément du tableau A_{k+1} sera calculé en fonction d'un patron de données du tableau A_k et l'une de ces données est produite par l'itération $\vec{l}$.

Afin d'exploiter ces registres vivants dans les prochaines itérations qui suivent $\vec{l}$, nous devons allouer $\prod_{i=1}^{n} T_i^r + Nbr$ registres :

1. $\prod_{i=1}^{n} T_i^r$ est le nombre de données du tableau A_k que nous produisons à chaque nouvelle itération $\vec{l}$ du code en *figure 5.8* (*page 96*) ;
2. Nbr est le nombre de registres vivants.

Définition 34 (Registre vivant) *Soit $\vec{l} = (\vec{l}^{\mathcal{E}},\ \vec{l}^{\mathcal{R}})^t$ une itération du code en* figure 5.8 *(*page 96*). Un registre qui contient une donnée du tableau A_k est dit vivant à l'itération $\vec{l}$ si et seulement si :*

- *la donnée du tableau A_k qu'il contient est produite par une itération déroulée $\vec{l'}$ telle que : $l'_1 = l_1, \cdots, l'_{2n-1} = l_{2n-1}$ et $l'_{2n} < l_{2n}$*
- *il existe une itération déroulée $\vec{l}''$ qui va la consommer telle que $l_1^{''} = l_1, \cdots,\ l_{2n-1}^{''} = l_{2n-1}$ et $l_{2n}^{''} \geq l_{2n}$,*

Soit $\vec{l} = (\vec{l}^{\mathcal{E}},\ \vec{l}^{\mathcal{R}})^t$ une itération du code en *figure 5.8* (*page 96*). Le nombre de registres qui contiennent des données du tableau A_k et qui sont vivants à l'itération $\vec{l}$ est borné par :

$$d_n^{k,\ max,\ c} * \prod_{i=1}^{n-1} \mathcal{R}_i \vec{T}^r$$

où $d_n^{k,\ max,\ c}$ est la plus grande projection de toutes les dépendances sur l'axe m (m est le numéro de la colonne de la matrice $\mathcal{R}$ telle que $\mathcal{R}_{n,m} = 1$) qui ont une projection sur chacun des autres axes inférieure à la taille des tuiles internes sur cet axe.

5.3.10 Génération du code dans le cas général

1) Gestion des tampons : De la même manière que la cas à deux dimensions, pour éviter la copie du tampon $B'_{k,i}$ $(1 \leq i \leq n)$ dans le tampon $B_{k,i}$, nous fusionnons les tampons $B_{k,i}$ et $B'_{k,i}$ $(i = 1...n)$ en un seul tampon qui sera noté par $BT_{k,i}[2]$ tel que $|BT_{k,i}[0]| = |BT_{k,i}[1]| = |B_{k,i}| = |B'_{k,i}|$. Dans ce cas, une itération $\vec{l}$ qui doit stocker une donnée du tableau A_k dans le tampon $B'_{k,i}$ la stocke dans $BT_{k,i}[l_1 \bmod 2]$ et si elle doit récupérer une donnée dans le tampon $B_{k,i}$, elle va la récupérer dans $BT_{k,i}[(l_1 - 1) \bmod 2]$.

2) Gestion des registres : Soit $\vec{l} = (\vec{l^{\mathcal{E}}},\ \vec{l^{\mathcal{R}}})^t$ une itération dans le code *figure 5.8* (*page 96*). Les $\prod_{i=1}^{n} T_i^r$ éléments du tableau A_k produits à cette itération seront sauvegardés dans $\prod_{i=1}^{n} T_i^r$ registres différents. Comme nous l'avons mentionné précédemment, en plus de ces registres, nous avons besoin de $d_2^{k,max,c} * \prod_{i=1}^{n-1} \mathcal{R}_i \vec{T}^r$ registres supplémentaires qui contiendront les données du tableau A_k :

1. qui sont produites par des itérations $\vec{l'} \prec \vec{l}\,(l'_1 = l_1, \cdots, l_{2n-1} = l'_{2n-1}$ et $l'_{2n} < l_{2n})$;
2. qui sont susceptibles d'être utilisées par des itérations $\vec{l}'' \succeq \vec{l}\,(l_1'' = l_1, \cdots, l_{2n-1}'' = l_{2n-1}$ et $l_{2n}'' \geq l_{2n})$.

Ces données sont produites dans $m = \lceil \frac{d_2^{k,max,c}}{\mathcal{R}_n \vec{T}^r} \rceil$ itérations différentes (le nombre d'itérations $\vec{l'}$ possibles). Ces différentes itérations seront notées par :

$$\vec{l}^1, \cdots, \vec{l}^m \ \text{ avec } \vec{l}^1 \prec \cdots \prec \vec{l}^m$$

Nous pouvons vérifier facilement que :

1. $\forall i \in [2..m]$: $\vec{l}^i$ produit $\prod_{i=1}^{n} T_i^r$ éléments du tableau A_k ;
2. $\vec{l}^1$ produit $\begin{cases} \prod_{i=1}^{n} T_i^r \text{ éléments} & \text{si } d_2^{k,max,c} \bmod (\mathcal{R}_n \vec{T}^r) = 0 \\ (d_2^{k,max,c} \bmod (\mathcal{R}_n \vec{T}^r)) * \prod_{i=1}^{n-1} \mathcal{R}_n \vec{T}^r \text{ éléments} & \text{sinon.} \end{cases}$

Nous notons l'itération $\vec{l}$ par $\vec{l}^{m+1}$ et l'itération qui la suit par $\vec{l}^{m+2}$, c'est-à-dire que $\vec{l}^{m+2} = \vec{l} + (0, \cdots,\ 0,\ 1)^t$. Lorsque nous passons de l'itération $\vec{l}$ vers $\vec{l}^{m+2}$, nous devons mettre à jour les différents registres de la manière suivante :

```
si d_2^{k,max,c} mod (R_n T^r) ≠ 0 alors
    les registres associés à l^1 vont recevoir une partie des registres associés à l^2
sinon
    les registres associés à l^1 vont recevoir les registres associés à l^2
finsi
pour i = 2 à i = m
    les registres associés à l^i vont recevoir les registres associés à l^{i+1}
finpour
```

Figure 5.9:

3) Écriture et lecture des éléments des différents tableaux A_k : Comme nous l'avons mentionné précédemment, à chaque itération déroulée $\vec{l}$, nous calculons un nouvel élément du tableau A_{k+1} en fonction d'un patron de données du tableau A_k. Ce nouvel élément du tableau A_{k+1} sera toujours rangé dans le tampon $B_{k,n+1}$ et dans l'un des $\prod_{i=1}^{n} T_i^r$ registres associés à $\vec{l}$. Par contre il sera rangé dans le tampon $B'_{k,i}$ $(1 \leq i \leq n)$ si $T_1^r * l_{n+i} + l_{2n+i} \geq \mathcal{E}_i \vec{T}^e - \mathcal{E}_i \vec{d}^{k,max}$.

Un élément du tableau A_k nécessaire pour le calcul de l'élément du tableau A_{k+1} à l'itération $\vec{l}$ sera récupéré dans l'un des registres qui sont associés ou vivants à l'itération $\vec{l}$ ou dans l'un des tampons $B_{k,i}$ $(1 \leq i \leq n)$. Soit $\vec{l'}$ l'itération déroulée qui produit cet élément du tableau A_k :

- si $l'_1 = l_1 - 1$, alors il sera récupéré dans le tampon $B_{k,1}$;
- si $l'_1 = l_1$ et $l'_2 = l_2 - 1$, alors il sera récupéré dans le tampon $B_{k,2}$;

- .
- .
- si $l'_1 = l_1, \cdots, l'_{m-1} = l_{m-1}$ et $l'_m = l_m - 1$, alors il sera récupéré dans le tampon $B_{k,m}$;
- .
- .
- si $l'_1 = l_1, \cdots, l'_{n-1} = l_{n-1}$ et $l'_n = l_n - 1$ alors il sera récupéré dans le tampon $B_{k,n}$;
- si $l'_1 = l_1, \cdots, l'_n = l_n$ et $(l'_{n+1}, \cdots, l'_{2n-1})^t \prec (l_{n+1}, \cdots, l_{2n-1})^t$ alors il sera récupéré dans le tampon $B_{k,n+1}$;
- si $l'_1 = l_1, \cdots, l'_n = l_n$, $(l'_{n+1}, \cdots, l'_{2n-1})^t = (l_{n+1}, \cdots, l_{2n-1})^t$ et $l'_{2n} < l_{2n}$ alors il sera récupéré dans l'un des registres qui sont vivants à l'itération $\vec{l}$;
- si $l'_1 = l_1, \cdots, l'_n = l_n$, $(l'_{n+1}, \cdots, l'_{2n-1})^t = (l_{n+1}, \cdots, l_{2n-1})^t$ et $l'_{2n} = l_{2n}$ alors il sera récupéré dans l'un des registres qui sont associés à l'itération $\vec{l}$.

5.4 Conclusion

Comme nous l'avons signalé dans le chapitre précédent, la plupart des travaux qui sont consacrés au pavage traitent un seul niveau de la hiérarchie mémoire. Dans ce chapitre, nous avons appliqué la fusion avec pavage à deux niveaux sur une séquence de nids de boucles. Le premier niveau est destiné au cache externe et le deuxième niveau est destiné aux registres. Grâce à notre modélisation, nous pouvons étendre facilement notre pavage à plus de deux niveaux ; il suffit de changer les matrices $\mathcal{A}$ et $\mathcal{P}$. Nous avons donné deux solutions pour les vecteurs de décalage pour que notre pavage avec fusion soit valide. Ensuite, nous avons développé une nouvelle méthode d'allocation des tampons et des registres. Cette méthode divise l'ensemble des données vivantes de chaque tableau en plusieurs sous-ensembles et attribue à l'un de ces sous-ensembles un ensemble de registres et à chacun des autres un tampon. Pour ces différents tampons, nous avons défini des fonctions d'accès qui permettent de ranger et de récupérer les données du tableau correspondant et nous avons pris en considération la réutilisation des registres lorsque nous passons d'une tuile à la suivante.

Chapitre 6

Élimination des gardes dans les transformées unimodulaires

Plusieurs travaux [80, 5, 43, 76] ont utilisé les transformations unimodulaires pour augmenter la localité des données, augmenter le degré du parallélisme, ...etc. Cependant la plupart sont destinés aux nids de boucles parfaitement imbriquées. Dans la pratique, les programmes contiennent surtout des nids de boucles mal imbriquées, et leur traitement pourraient permettre obtenir des gains considérables. C'est pour ces raisons que nous nous sommes intéressés au problème de l'élimination des gardes au sein de nids de boucles.

L'application d'une transformation unimodulaire directement sur un nid de boucles mal imbriquées n'est pas possible. Nous sommes donc amenés à rendre ce dernier parfaitement imbriqué en ajoutant des gardes sur les instructions que nous faisons rentrer dans la boucle la plus interne avant de pouvoir lui appliquer une transformation unimodulaire. Wolf [75] et Abu-Sufah [1] ont mentionné l'utilisation de cette technique mais n'ont pas envisagé tous les cas possibles et ils ont signalé que les gardes ajoutées consomment beaucoup de temps supplémentaires dans le code transformé. Wolfe [79] a par contre proposé d'appliquer d'abord une distribution de boucles, mais celle-ci n'est pas toujours possible et crée des problèmes de performance. Xue [83] a proposé un algorithme de génération de code sans gardes qui est différent de notre algorithme.

Dans ce chapitre, nous sommes intéressés à la condition nécessaire et suffisante pour que la transformation d'un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées soit correcte. Nous avons aussi traité le cas où cette condition n'est pas vérifiée. En deuxième lieu, nous donnons un algorithme qui permet d'éliminer les gardes dans le code transformé en découpant ce dernier en une séquence de nids de boucles. Chamski [21] a aussi proposé d'utiliser le découpage de boucles pour éliminer les calculs de maxima et de minima dans les bornes de boucles. Par contre, Lengauer [53] et Wetzel [74] se sont intéressés aux techniques de fusion et de distribution de boucles pour la génération de code.

Ce chapitre est organisé de la manière suivante. Nous commençons par décrire le problème avec un simple exemple, le produit matrice/vecteur. Puis nous donnons la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées. Ensuite nous décrivons l'algorithme qui permet d'éliminer les gardes dans le nid transformé. Ces deux algorithmes sont ensuite comparés aux travaux antérieurs. Enfin, nous terminons par une conclusion récapitulant les résultats et présentant les travaux futurs.

6.1 Exemple introductif

Afin de mieux comprendre le problème traité, nous donnons un simple exemple qui est le produit matrice/vecteur, dont l'algorithme est présenté en *figure 6.1(a)*.

```
do ( i = 0, i = 4 )
   S = 0
   do (j = 0 , j = 6 )
      S = S + A(i,j) * X(j)
   enddo
   Y(i) = S
enddo
(a)
```

```
do ( i = 0, i = 4 )
   do (j = 0 , j = 6 )
      if (j = 0) then S = 0
      S = S + A(i,j) * X(j)
      if (j = 6) then Y(i) = S
   enddo
enddo
(b)
```

Figure 6.1: Produit matrice/vecteur : passage du nid mal imbriqué au nid parfaitement imbriqué

Pour appliquer une transformation unimodulaire sur le code en *figure 6.1(a)*, nous devons tout d'abord le rendre parfaitement imbriqué, c'est-à-dire transférer les instructions `S=0` et `Y(i)=S` à l'intérieur de la boucle `j`. eci est possible car la condition nécessaire et suffisante est vérifiée : pour toute valeur de i, il existe au moins une itération j. Le nid de boucles parfaitement imbriquées correspondant au code de la *figure 6.1(a)* est donné en *figure 6.1(b)*.

Une transformation unimodulaire est une transformation affine. Elle transforme donc une droite qui est à la frontière d'un polytope initial en une droite qui est à la frontière du polytope transformé.

Par exemple, la transformation unimodulaire du code en *figure 6.1(b)* par $T = \begin{pmatrix} 6 & 1 \\ 1 & 0 \end{pmatrix}$, donne le code en *figure 6.2*.

```
do ( i' = 0, i' = 30 )
   do ( (−j' ≤ 0) ∧ (i' − 6 * j' ≤ 6), (j' ≤ 4) ∧ (−i' + 6 * j' ≤ 0))
      if (−i' + 6 * j' = 0) then S = 0
      S = S + A(j', i' − 6 * j') * X(i' − 6 * j')
      if (i' − 6 * j' = 6) then Y(j') = S
   enddo
enddo
```

Figure 6.2: Algorithme transformé

L'image de la droite définie par $j = 0$ qui est à la frontière du polytope définissant le domaine d'itérations du code en *figure 6.1(b)* est la droite définie par $-i' + 6 * j' = 0$ qui est à la frontière du polytope du domaine d'itérations du code en *figure 6.2*.

Lorsque nous transformons un nid mal imbriqué en un nid parfaitement imbriqué, nous ajoutons des gardes sur les instructions que nous faisons rentrer dans la boucle la plus interne. Ces gardes sont définies par des droites qui sont aux frontières du polytope du domaine initial. Par conséquent, dans le nid transformé, les images de ces gardes correspondent à des droites qui sont toujours aux frontières du polytope correspondant.

La deuxième étape consiste à éliminer les gardes dans le code transformé. Pour pouvoir réaliser cela, nous décomposons le nid de boucles en *figure 6.2* en une séquence de nids de boucles de manière à avoir dans chaque séquence une seule contrainte inférieure et une seule contrainte supérieure dans la boucle interne j'. Cette décomposition se fait en utilisant les sommets générateurs du polytope transformé et revient à transformer l'unique nid en *figure 6.2* en une séquence de la *figure 6.4*.

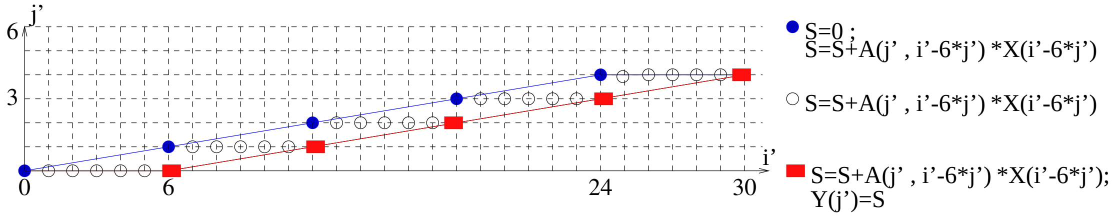


Figure 6.3: Domaine d'itérations du code en *figure 6.2*

L'étude du code transformé en *figure 6.2* et du domaine d'itérations en *figure 6.3* montre que nous pouvons extraire l'affectation `S=0`, gardée par la condition $-i' + 6 * j' = 0$, après la boucle j', car elle s'exécute uniquement au cours de la dernière itération de j'. En effet, la droite définie par $-i' + 6 * j' = 0$ est la frontière haute du domaine d'itérations. L'affectation `S=0` s'exécute avant l'affectation :

S=S+A(j',$i' - 6 * j'$) * X($i' - 6 * j'$)

On extrait ces deux instructions de la boucle j' et on les garde par la condition $i' \mod 6 = 0$, car elles ne s'exécutent que pour des valeurs de i' qui sont multiples de 6.

Par contre l'affectation $Y(j') = S$, gardée par $i' - 6 * j' = 6$, sera extraite avant la boucle j', car elle s'exécute uniquement au cours de la première itération de j'. En effet la droite définie par $i' - 6 * j' = 6$ est la frontière inférieure du domaine d'itération. l'affectation $Y(j') = S$ s'exécutant après l'affectation :

S=S+A(j',$i' - 6 * j'$) * X($i' - 6 * j'$))

On extrait ces deux instructions de la boucle j' et on les garde par la condition $(i' - 6)$ `mod 6=0`, car elles ne s'exécutent que pour des valeurs de i' telles que $(i' - 6)$ est multiple de 6.

```
do ( i' = 0, i' = 5 )
   do ( j' = 0 , j' = i'/6 - 1/6)
      S = S + A(j', i' - 6 * j') * X(i' - 6 * j')
   enddo
   if (entier i'/6) then { S = 0 ; S = S + A(i'/6, 0) * X(0) }
enddo
do ( i' = 6, i' = 24 )
   if (entier (i'-6)/6) then { S = S + A((i'-6)/6, 6) * X(6) ; Y(((i'-6)/6) = S}
   do ( j' = i/6 , j' = i'/6 - 1/6)
      S = S + A(j', i' - 6 * j') * X(i' - 6 * j')
   enddo
   if (entier i'/6) then { S=0 ; S = S + A(i'/6, 0) * X(0) }
enddo
do ( i' = 25, i' = 30 )
   if (entier (i'-6)/6) then { S = S + A((i'-6)/6, 6) * X(6); Y((i'-6)/6) = S }
   do ( j' = i/6 , j' = 4)
      S = S + A(j', i' - 6 * j') * X(i' - 6 * j')
   enddo
enddo
```

Figure 6.4: Code transformé et décomposé

6.2 Transformation en nid de boucles parfaitement imbriquées

Les nids de boucles que nous traitons dans ce chapitre ont la forme du code donné en *figure 6.5(a)*. Avant de leur appliquer une transformation unimodulaire, nous devons les mettre sous la forme du code donné en *figure 6.5(b)*. Ceci est possible si et seulement si la *proposition 23* (*page 105*) est vérifiée. Avant de démontrer la *proposition 23*, nous introduisons la *proposition 22* (*page 105*).

```
do (i1 = inf, i1 = sup)
   S1
   do (i2 = a1 * i1 + b1 , i2 = a2 * i1 + b2 )
      S2
   enddo
   S1'
enddo
```

(a)

```
do (i1 = inf, i1 = sup)
   do (i2 = a1 * i1 + b1 , i2 = a2 * i1 + b2 )
      If (i2 = a1 * i1 + b1) then S1
      S2
      If (i2 = a2 * i1 + b2) then S1'
   enddo
enddo
```

(b)

Figure 6.5: Programmes traités

Proposition 22 *Soit inf et sup deux entiers tels que $inf \leq sup$ alors nous avons l'équivalence suivante :*

$$\begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases} \iff \begin{cases} (a_1 = a_2) \wedge (b_2 \geq b_1) \\ \qquad \vee \\ (a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \\ \qquad \vee \\ (a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1) \end{cases}$$

Preuve :

$\Rightarrow$? Nous supposons que $\begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases}$.

Nous distinguons les 3 cas :

1. $a_1 = a_2$ alors nécessairement $b_2 \geq b_1$.
2. $a_1 > a_2$, nous avons par supposition $(a_2 * sup + b_2 \geq a_1 * sup + b_1)$.
3. $a_1 < a_2$, nous avons aussi par supposition $(a_2 * inf + b_2 \geq a_1 * inf + b_1)$.

$\Leftarrow$? Nous supposons que $\begin{cases} (a_1 = a_2) \wedge (b_2 \geq b_1) \\ \qquad \vee \\ (a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \\ \qquad \vee \\ (a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1) \end{cases}$

1. $(a_1 = a_2) \wedge (b_2 \geq b_1) \Longrightarrow \begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases}$
2. $(a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \Longrightarrow (a_1 - a_2) * sup \leq b_2 - b_1$. Comme $(a_1 - a_2)$ est positif et $sup \geq inf$ alors $(a_1 - a_2) * inf \leq b_2 - b_1 \Longrightarrow a_2 * inf + b_2 \geq a_1 * inf + b_1$.
3. $(a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1)$ alors $(a_2 - a_1) * inf \geq b_1 - b_2$. Comme $(a_2 - a_1)$ est positif et $inf \leq sup$ alors $(a_2 - a_1) * sup \geq b_1 - b_2 \Longrightarrow (a_2 * sup + b_2 \geq a_1 * sup + b_1)$.

$\square$

Proposition 23 *Le code en* figure 6.5(a) *et le code en* figure 6.5(b) *sont équivalents si et seulement si, les deux conditions suivantes sont vérifiées :*

$$\begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases}$$

Preuve :

1) $\Longrightarrow$?

Le code en *figure 6.5(a)* et le code en *figure 6.5(b)* sont équivalents $\Longrightarrow$

– $\forall\ i_1 \in [inf, sup], \exists!\ i_2 = a_1 * i_1 + b_1\ /\ e(S_1(i_1, i_2))$ sera exécutée.

– $\forall\ i_1 \in [inf, sup], \exists!\ i_2 = a_2 * i_1 + b_2 /\ e(S'_1(i_1, i_2))$ sera exécutée.

L'unicité est assurée par le fait que $i_2 = a_1 * i_1 + b_1$ et $i_2 = a_2 * i_1 + b_2$ sont des fonctions linaires affines et injectives.

L'existence est assurée si et seulement si le domaine défini par

$$\left\{\begin{array}{l} inf \leq i_1 \leq sup \\ a_1 * i_1 + b_1 > a_2 * i_1 + b_2 \end{array}\right. = \left\{\begin{array}{l} inf \leq i_1 \leq sup \\ \\ (a_1 - a_2) * i_1 > b_2 - b_1 \end{array}\right. \quad (I) \quad \text{est vide.}$$

Cas 1 $a_1 = a_2$ **:** Le domaine défini par (I) n'admet aucune solution si est seulement si $b_2 \geq b_1$.

Cas 2 $a_1 > a_2$ **:** Le domain définie par (I) n'admet aucune solution $\Longleftrightarrow \frac{b_2-b_1}{a_1-a_2} \geq sup$ c'est à dire $a_2 * sup + b_2 \geq a_1 * sup + a_1$.

Cas 3 $a_1 < a_2$ **:** le domaine défini par (I) n'admet aucune solution $\Longleftrightarrow \frac{b_2-b_1}{a_1-a_2} \leq inf$ c'est à dire $a_2 * inf + b_2 \geq a_1 * inf + a_1$.

Donc le code en *figure 6.5(a)* est équivalent au code en *figure 6.5(b)* $\Longrightarrow$

$$\left\{\begin{array}{l} (a_1 = a_2) \wedge (b_2 \geq b_1) \\ \qquad \vee \\ (a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \quad (II) \\ \qquad \vee \\ (a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1) \end{array}\right.$$

D'après la *proposition 22* (*page 105*) et (II) le code en *figure 6.5(a)* est équivalent au code en *figure 6.5(b)*, alors :

$$\left\{\begin{array}{l} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{array}\right.$$

2) $\Longleftarrow$**?**

$\left\{\begin{array}{l} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{array}\right. \Longrightarrow \forall i_1 \in [inf, sup] : a_1 * i_1 + b_1 \leq a_2 * i_1 + b_2$ car les fonctions sont monotones : $\Longrightarrow$

$$\left\{\begin{array}{l} \forall i_1 \in [inf, sup], \exists !\ i_2 = a_1 * i_1 + b_1 /\ e(S_1(i_1, i_2)) \ \text{ sera exécutée} \\ \forall i_1 \in [inf, sup], \exists !\ i_2 = a_2 * i_1 + b_2 / e(S'_1(i_1, i_2)) \ \text{ sera exécutée} \end{array}\right.$$

Donc les nombres d'occurences de S_1 dans le code en *figure 6.5(a)* et le code en *figure 6.5(b)* sont égaux, de même pour S'_1. Et on peut vérifier facilement que l'ordre d'exécution des deux programmes est le même. Par conséquent le code en *figure 6.5(a)* et le code en *figure 6.5(b)* sont équivalents.

□

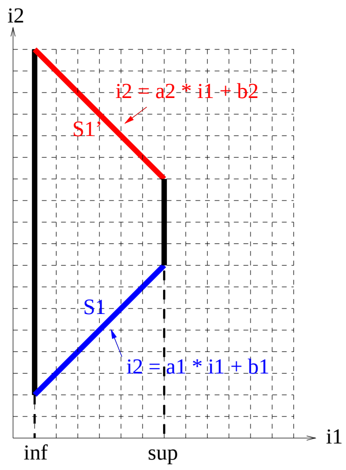
6.2.1 Décomposition en séquence de nids de boucles

D'après la *proposition 23* (*page 105*), le code en *figure 6.5(a)* peut être mis sous 4 formes, suivant la position des droites définies par $i_2 = a_1 * i_1 + b_1$ et $i_2 = a_2 * i_1 + b_2$. Nous posons $f_1(i_1) = a_1 * i_1 + b_1$ et $f_2(i_1) = a_2 * i_1 + b_2$.

1. Cas 1 : $f_2(inf) \geq f_1(inf) \wedge f_2(sup) \geq f_1(sup)$
   Le code en *figure 6.5(a)* sera mis sous la forme du code en *figure 6.6(a)*.

do (i_1 = inf, i_1 = sup)
    do ($i_2 = a_1 * i_1 + b_1$, $i_2 = a_2 * i_1 + b_2$)
        If $i_2 = a_1 * i_1 + b_1$ then S_1
        S_2
        If $i_2 = a_2 * i_1 + b_2$ then S'_1
    enddo
enddo

(a)



(b)

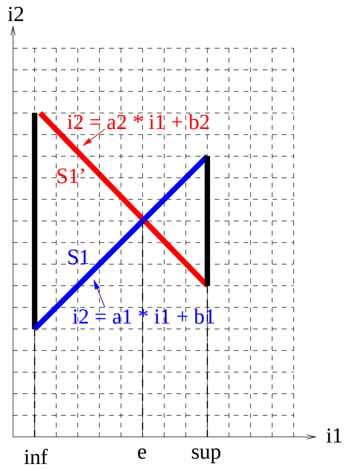
Figure 6.6: Cas 1

La *figure 6.6(b)* illustre un exemple qui entre dans le cas 1 où $f_1(i_1) = i_1 + 1$ et $f_2(i_1) = -i_1 + 19$.

2. Cas 2 : $f_2(inf) \geq f_1(inf) \wedge f_2(sup) < f_1(sup)$
   Soit e un réel tel que $f_1(e) = f_2(e)$, alors le code en *figure 6.5(a)* sera mis sous la forme du code en *figure 6.7(a)*.

do (i_1 = inf, i_1 = $\lfloor e \rfloor$)
    do ($i_2 = a_1 * i_1 + b_1$, $i_2 = a_2 * i_1 + b_2$)
        If $i_2 = a_1 * i_1 + b_1$ then S_1
        S_2
        If $i_2 = a_2 * i_1 + b_2$ then S'_1
    enddo
enddo
do (i_1 = $\lfloor e \rfloor$+1 , i_1 = sup)
    S_1
    S'_1
enddo

(a)



(b)

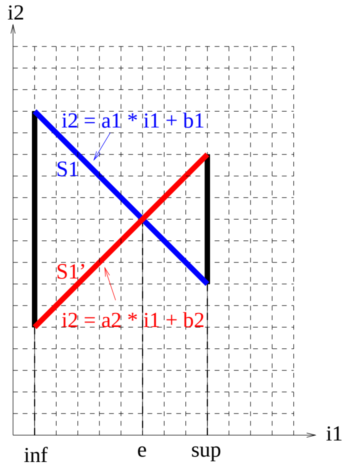
Figure 6.7: Cas 2

La *figure 6.7(b)* illustre le cas où $f_1(i_1) = i_1 + 4$ et $f_2(i_1) = -i_1 + 16$. Pour des valeurs de i_1 supérieure à $\lfloor e \rfloor$, il n'y a plus existence de valeurs pour i_2 d'où l'élimination lors de la génération de code du nid de boucle interne.

3. Cas 3 : $f_2(inf) < f_1(inf) \wedge f_2(sup) \geq f_1(sup)$
   Soit e un réel tel que $f_1(e) = f_2(e)$, alors le code en *figure 6.5(a)* sera mis sous la forme du code en *figure 6.8(a)*.

```
do (i1 =inf , i1 = ⌈e⌉-1)
    S1
    S1'
enddo
do (i1 = ⌈e⌉, i1 = sup)
    do (i2 = a1 * i1 + b1 , i2 = a2 * i1 + b2 )
        If i2 = a1 * i1 + b1 then S1
        S2
        If i2 = a2 * i1 + b2 then S1'
    enddo
enddo
```

(a)



(b)

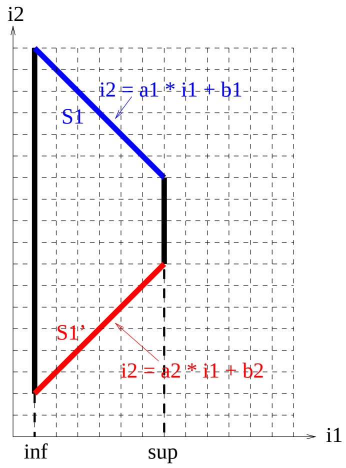
Figure 6.8: Cas 3

La *figure 6.8(b)* illustre le cas où $f_1(i_1) = -i_1 + 16$ et $f_2(i_1) = i_1 + 4$. Pour des valeurs de i_1 inférieure à $\lceil e \rceil$, il n'y a plus existence de valeurs pour i_2 d'où l'élimination lors de la génération de code du nid de boucle interne.

4. Cas 4 : $f_2(inf) < f_1(inf) \wedge f_2(sup) < f_1(sup)$
   Le code en *figure 6.5(a)* sera mis sous la forme du code en *figure 6.9(a)*.

```
do (i1 = inf, i1 = sup)
    S1
    S1'
enddo
```

(a)



(b)

Figure 6.9: Cas 4

La *figure 6.9(b)* illustre le cas où $f_1(i_1) = -i_1 + 19$ et $f_2(i_1) = i_1 + 1$. Pour toute valeur de i_1, il n'y a plus existence de valeurs pour i_2 d'où l'élimination lors de la génération de code du nid de boucle interne.

6.3 Elimination des gardes dans le nid de boucle transformé

Cette section présente les techniques que nous proposons d'utiliser pour générer une séquence de nids de boucles associée au code transformé qui soit le plus simple possible, c'est-à-dire comportant le moins de gardes possibles.

Après l'application de la transformation unimodulaire T, le nid de boucles aura la forme du code en *figure 6.10*.

do $(-i'_1 \leq -b_0,\ i'_1 \leq b'_0)$
  do $(\wedge_{j=1}^{n}(a_j * i'_1 - d_j * i'_2 \leq b_j),\ \wedge_{j=1}^{n'}(a'_j * i'_1 + d'_j * i'_2 \leq b'_j\ ))$
    If $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ alors S_1
    S_2
    If $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ alors S'_1
  enddo
enddo

Figure 6.10: Programme après transformation unimodulaire

Les droites définies par $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ et $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ sont toujours aux frontières du polytope du code en *figure 6.10*; elles peuvent être soit sur la frontière inférieure, soit sur la frontière supérieure, soit perpendiculaires à l'axe i'_1.

La borne inférieure de la boucle sur i'_2 peut être composée d'une ou deux contraintes, de même pour la contrainte supérieure. Le nombre de structures (nids de boucles) possibles que peut prendre le code en *figure 6.10* après l'élimination des gardes est très important, car le programme résultant dépend de la position des droites définies par $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ et $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ et du nombre de contraintes inférieures et supérieures de la boucle i'_2.

Nous voulons ramener le code en *figure 6.10* sous la forme du code en *figure 6.11*.

do $(-i'_1 \leq -b_0,\ i_1 \leq b'_0)$
  S_1
  do $(\wedge_{j=1}^{n}(a_j * i'_1 - d_j * i'_2 \leq b_j),\ \wedge_{j=1}^{n'}(a'_j * i'_1 + d'_j * i'_2 \leq b'_j\ ))$
    S_2
  enddo
  S'_1
enddo

Figure 6.11: Programme sans gardes

Proposition 24 *Le code de la* figure 6.10 *et celui de la* figure 6.11 *sont équivalents si et seulement si :*

$$\begin{cases} n = n' = 1 \\ (h_{m,1} = a_j) \wedge (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \\ (h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \end{cases}$$

Preuve :

$\Rightarrow$? Les codes en *figure 6.10* et en *figure 6.11* sont équivalents alors :

(a) A toute occurence $g(S_1(i'_1))$ doit correspondre une et une seule occurence $f(S_1(i'_1, i'_2))$ c'est à dire $\forall i'_1, \exists! i'_2 /\ i'_2 = \frac{-b_m + h_{m,1} * i'_1}{-h_{m,2}}$ => $h_{m,2} = 1$ *ou* $h_{m,2} = -1$.

(b) Pour tout i'_1, l'occurence $f(S_1(i'_1, i'_2 = \frac{-b_m+h_{m,1}*i'_1}{-h_{m,2}}))$ doit être exécutée avant la première occurence $f(S_2(i'_1, i'_2 = \frac{-b_j+a_j*i'_1}{d_j}))$, c'est à dire $\forall i'_1, \forall j$: $f(S_1(i'_1, i'_2 = \frac{-b_m+h_{m,1}*i'_1}{-h_{m,2}})) <_E f(S_2(i'_1, i'_2 = \frac{-b_j+a_j*i'_1}{d_j}))$. Ceci est possible si et seulement si quelque soit j : $\frac{-b_m+h_{m,1}*i'_1}{-h_{m,2}} = \frac{-b_j+a_j*i'_1}{d_j}$, c'est à dire $\frac{-b_1+a_1*i'_1}{d_1} = ... = \frac{-b_n+a_n*i'_1}{d_n}$. Comme les d_j sont positifs, alors nécessairement n=1.

Comme $\frac{-b_m+h_{m,1}*i'_1}{-h_{m,2}} = \frac{-b_j+a_j*i'_1}{d_j}$ alors on distingue deux cas :

Cas 1 : $-h_{m,2} = d_j$ *et* $\forall i'_1$: $(-b_j + a_j * i'_1) = (-b_m + h_{m,1} * i'_1)$, c'est à dire $h_{m,2} = -d_j$, $b_m = b_j$ *et* $h_{m,1} = a_j$

Cas 2 : $h_{m,2} = d_j$ *et* $\forall i'_1$: $-(-b_j + a_j * i'_1) = (-b_m + h_{m,1} * i'_1)$, c'est à dire $h_{m,2} = d_j$, $b_m = -b_j$ *et* $h_{m,1} = -a_j$

Les Cas 1 et Cas 2 sont équivalents car $(h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m)$ est équivalent à $(-h_{m,1} * i'_1 - h_{m,2} * i'_2 = -b_m)$. Comme d_j est positif, $h_{m,2}= -d_j$ et $h_{m,2} = -1$ alors $d_j = 1$. En résumé, les codes en *figure 6.10* et en *figure 6.11* sont équivalents alors :

$$\begin{cases} n = 1 \\ (h_{m,1} = a_j) \wedge (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \end{cases}$$

De la même manière on démontre que si les codes en *figure 6.10* et en *figure 6.11* sont équivalents alors :

$$\begin{cases} n' = 1 \\ (h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \end{cases}$$

$\Leftarrow$? Si on a $\begin{cases} n = n' = 1 \\ (h_{m,1} = a_j) \wedge (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \\ (h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \end{cases}$

Alors le code en *figure 6.10* va se mettre sous la forme du code en *figure 6.12*

```
do (−i'_1 ≤ −b_0, i'_1 ≤ b'_0)
    do (a_1 ∗ i'_1 − i'_2 ≤ b_1, a'_1 ∗ i'_1 + i'_2 ≤ b'_1 )
        If (a_1 ∗ i'_1 − i'_2 = b_1) alors S_1
        S_2                 (d)
        If (a'_1 ∗ i'_1 + i'_2 = b'_1) alors S'_1
    enddo
enddo
```

Figure 6.12: Programme avec gardes

et le code *figure 6.11* va se mettre sous la forme donnée en *figure 6.13*.
A tout point (i_1, i_2) de l'espace initial correspond un point (i'_1, i'_2) dans l'espace image => il n'existe pas de i'_1 tel que le domaine de i'_2 soit vide alors $(b'_1 - a'_1 * b'_0) \geq (a_1 * b'_0 - b_1)$ et $(b'_1 - a'_1 * b_0) \geq (a_1 * b_0 - b_1)$. D'après la *proposition 23* (*page 105*) les codes en *figure 6.10* et en *figure 6.11* sont équivalents.

□

```
do (-i'_1 ≤ -b_0, i'_1 ≤ b'_0)
   S_1
   do (a_1 * i'_1 - i'_2 ≤ b_1, a'_1 * i'_1 + i'_2 ≤ b'_1 )
      S_2                (c)
   enddo
   S'_1
enddo
```

Figure 6.13: Programme sans gardes

6.3.1 Décomposition minimale du nid transformé

Malheureusement n et n' ne sont pas toujours égaux à 1. Par conséquent nous devons décomposer l'intervalle $[b_0, b'_0]$ en une union de sous intervalles tels que dans chaque intervalle on ait une seule contrainte inférieure $(a_j * i'_1 - d_j * i'_2 \leq b_j)$ et une seule contrainte supérieure $(a'_j * i'_1 + d'_j * i'_2 \leq b'_j)$.

Soit $G = \{ig_1, ig_2, ...ig_p\}$ l'ensemble des abscisses de tous les sommets générateurs du polytope du domaine transformé, trié en ordre croissant. Si deux sommets ont la même abscisse on ne prend qu'une seule valeur.

Soient les notations suivants :

1. $H_1^- = \{-i'_1 \leq -b_0\}$.

2. $H_1^+ = \{i'_1 \leq b'_0\}$.

3. $H_2^- = \{a_j * i'_1 - d_j * i'_2 \leq b_j,\ j = 1 : n\}$.

4. $H_2^+ = \{a'_j * i'_1 + d'_j * i'_2 \leq b'_j,\ j = 1 : n'\}$.

5. $C^- = \{c_{l,k}(i_1) = \frac{a_j * i'_1 - b_j + d_j - 1}{d_j}\ /\ (a_j * i'_1 - d_j * i'_2 \leq b_j) \in H_2^-\}$.

6. $C^+ = \{c'_{l,k}(i_1) = \frac{-a'_j * i'_1 + b'_j}{d'_j}\ /\ (a'_j * i'_1 + d'_j * i'_2 \leq b'_j) \in H_2^+\}$.

7. $\vec{j} = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$ le vecteur d'itération initial.

8. $\vec{j}' = \begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix}$ le vecteur d'itération après la transformation T.

9. $(h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m)$ est équivalent à $(h_m\ \vec{j}' = b_m)$.

10. $(h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b_{m'})$ est équivalent à $(h'_m\ \vec{j}' = b'_m)$.

Sachant que d_j et d'_j sont des entiers positifs non nuls.

Proposition 25 *Le découpage de l'intervalle* $[b_0,\ b'_0]$ *en* $[a_1,\ b_2] \cup [a_2,\ b_3] \cup [a_3,\ b_4] \cup [a_4,\ b_5].....\cup[a_{p-1},\ b_p]$ *tels que* $\left\{ \begin{array}{l} a_1 = ig_1 \\ b_p = ig_p \\ (b_i = ig_i \wedge a_i = ig_i + 1) ou (b_i = ig_i - 1 \wedge a_i = ig_i)\ pour\ i = 2, p-1 \end{array} \right.$

est le découpage minimal qui permet d'avoir une seule contrainte inférieure $(a_j * i'_1 - d_j * i'_2 \leq b_j)$ *et seule contrainte supérieure* $(a'_j * i'_1 + d'_j * i'_2 \leq b'_j)$ *dans chaque intervalle .*

Preuve :

1. Une seule contrainte supérieure et une seule contrainte inférieure dans chaque intervalle ?

   On suppose $\exists\ [a_i,\ b_{i+1}]$ tel que le nid de boucles lui correspondant contient plus d'une contrainte inférieure $(a_j{*}i'_1{-}d_j{*}i'_2 \leq b_j)$ ou plus d'une contrainte supérieure $(a'_j{*}i'_1{+}d'_j * i'_2 \leq b'_j)$ => $\exists$ un sommet générateur g_k tel que $ig_i < ig_k < ig_{i+1}$, ce qui représente une contradiction car l'ensemble G contient les abscisses de tous les sommets générateurs.

2. Découpage minimal ?

   On suppose qu'il existe un autre découpage qui contient moins d'intervalles => il existe un intervalle [a, b] tel que $\begin{cases} a \leq ig_i \ et \ b > ig_{i+1} \\ ou \\ a < ig_i \ et \ b \geq ig_{i+1} \end{cases}$. Dans cet intervalle, nécessairement on a soit plus d'une contrainte supérieure ou plus d'une contrainte inférieure, ce qui est contradictoire car on cherche des intervalles qui contiennent une seule contrainte supérieure et une seule contrainte inférieure.

□

6.3.2 Isolement d'un sommet générateur

Dans certains cas, illustrés en *figures 6.14*, *6.15* et *6.16*, il est nécessaire de générer un code spécifique correspondant à un sommet particulier du système de contraintes. Nous détaillons dans cette section les cas et les conditions correspondantes à ces générations de codes particuliers.

S'il existe un sommet générateur qui satisfait la *condition 1* (voir *figures 6.14*) :

$$Condition1 : [(h_m\ \vec{j}' = b_m) \cap (h_{m'}\ \vec{j}' = b_{m'}\ ){=}g_i] \wedge [\ (h'_m\ \vec{j}' \leq b'_m) \in H_2^-\ ],$$

alors le nid de boucles correspondant à l'intervalle $[ig_i, ig_i]$ sera traité comme un cas particulier. Dans tous les cas représentés en *figures 6.14*, le code S_1, S_2, S'_1 devra être extrait avant la boucle la plus interne.

Explication :

Prenons tout d'abord les exemples des *figures 6.14(a)* et *6.14(b)* qui satisfont la *condition 1*. Lors du processus de génération de code, on doit :

1. Extraire le code $S1'$ avant la deuxième boucle. Etant donné que l'ordre d'exécution est $S_1;\ S_2;\ S'_1$, on doit extraire aussi le code S_2.
2. Extraire le code $S1$ après la deuxième boucle. Pour la même raison que précédemment, on doit extraire aussi le code S_2.

1. et 2. conduisent à exécuter deux fois le code S_2 pour l'itération $i'_1 = ig_i$, ce qui génère un code faux car pour $i'_1 = ig_i$, le code S_2 n'est exécuté qu'une seule fois. Pour pallier ce problème on isole le nid de boucle correspondant à l'intervalle $[ig_i, ig_i]$ et on sort avant la

deuxième boucle le code S_1; S_2; S'_1, sachant que la deuxième boucle sera vide dans ces deux cas particuliers pour l'itération $i'_1 = ig_i$.

Dans les exemples des *figures 6.14(c)* et 6.14(d), on doit isoler le nid de boucles correspondant à l'intervalle $[ig_i, ig_i]$ car le code S_1 est exécuté uniquement pour $i'_1 = ig_i$. Comme le code S'_1 n'est exécuté que pour les premières itérations de i'_2 et qu'il s'exécute après S_1; S_2 alors on doit sortir le code S_1; S_2; S'_1 avant la deuxième boucle.

Dans les exemples des *figures 6.14(e)* et 6.14(f), on doit isoler le nid de boucles correspondant à l'intervalle $[ig_i, ig_i]$ car dans l'intervalle $[..., igi-1]$ on n'exécute uniquement que le code S_1 ; S_2 ou S_2 ; $S1'$. Dans l'intervalle $[ig_i, ig_i]$ on exécute le code S_1; S_2; S'_1 et dans l'intervalle $[ig_i+1, ...]$ on exécute uniquement le code S_2 ; S'_1 ou S_1 ; S_2. Comme les codes S_1 et S'_1 s'exécutent uniquement pour les premières itérations de i'_2, on extrait avant la deuxième boucle le code S_1; S_2; S'_1.

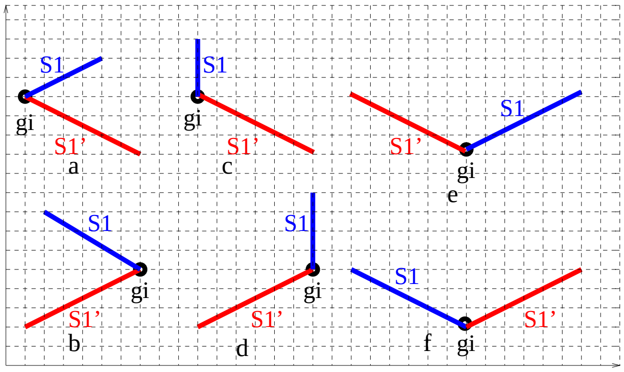


Figure 6.14: Les différents cas de la *condition 1*

De manière symétrique, si on a un sommet générateur qui satisfait la *condition 2* (voir *figures 6.15*) :

Condition2 : $[(h_m\ \vec{j}' = b_m) \cap (h'_m\ \vec{j}' = b'_m\ ){=}g_i] \wedge [(h_m\ \vec{j}' \leq b_m) \in H_2^+] \wedge \neg\ [(\ h'_m\ \vec{j}' \leq b'_m) \in H_2^-\ ]$,

alors on isole le nid de boucles correspondant à l'intervalle $[ig_i, ig_i]$ et on extrait le code S_1; S_2; S'_1 après la deuxième boucle.

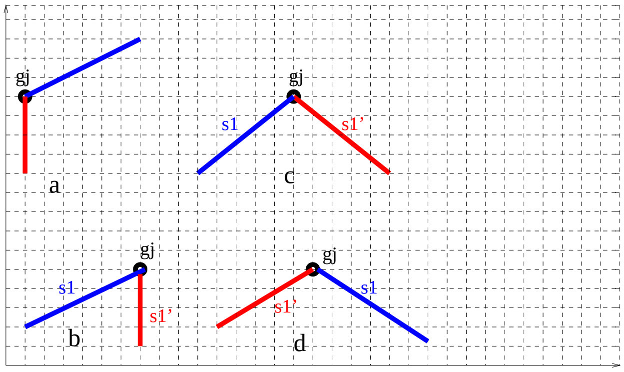


Figure 6.15: Les différents cas de la *condition 2*

Si on a un sommet générateur qui correspond à l'un des cas de la *figures 6.16* alors on doit isoler le nid de boucles correspondant à l'intervalle $[ig_i, ig_i]$. Mais dans ces derniers cas, il est possible de procéder pour l'élimination des gardes de la même manière que pour les autres intervalles.

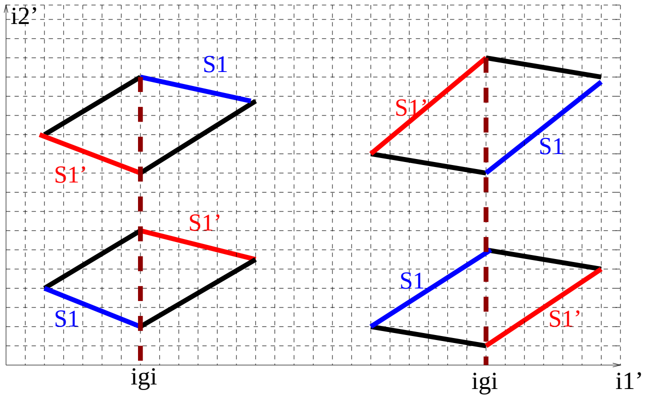


Figure 6.16: Cas particulier

6.3.3 Calcul des bornes de boucles et code généré

La *proposition 25* (*page 112*) et le traitement particulier de certains sommets pour lesquels on génère un nid de boucles spécifique conduisent au code intermédiaire donné en *figures 6.17*, avec :

1. $c_{j,j} = c_{l,k} \in C^-$ tel que $c_{l,k}(ig_j)$ est maximum. S'il y a deux fonctions $c_{l,k}$ qui satisfont cette condition (elles sont égales en ig_j) alors on prend un $c_{l,k} = \frac{a_j * i_1 - b_j + d_j - 1}{d_j}$ tel que $(a_j = h_{m,1} \wedge dj = -h_{m,2} \wedge b_j = b_m)$ ou $(a_j = h'_{m,1} \wedge dj = -h'_{m,2} \wedge b_j = b'_m)$.
2. $c'_{j,j} = c'_{l,k} \in C^+$ tel que $c'_{l,k}(ig_j)$ est minimum. S'il y a deux fonctions qui satisfont cette condition (elles sont égales en ig_j) alors on prend un $c'_{l,k} = \frac{-a'_j * i'_1 + b'_j}{d'_j}$ tel que $(a'_j = h_{m,1} \wedge dj' = h_{m,2} \wedge b'_j = b_m)$ ou $(a'_j = h'_{m,1} \wedge dj' = h'_{m,2} \wedge b'_j = b'_m)$.
3. $c_{j,j+1} = c_{l,k} \in C^-$ tel que $c_{l,k}(ig_j)$ et $c_{l,m}(ig_{j+1})$ sont maximum.
4. $c'_{j,j+1} = c'_{l,k} \in C^+$ tel que $c'_{l,k}(ig_j)$ et $c'_{l,m}(ig_{j+1})$ sont minimum.

```
do ( i'_1 = ig_1, i'_1 = ig_1 )
   do ( i'_2 = c_{1,1}, i'_2 = c'_{1,1})
      If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m then S_1
      S_2
      If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m then S'_1
   enddo
enddo
.
.
.
do ( i'_1 = ig_j, i'_1 = ig_j )
   do ( i'_2 = c_{j,j}, i'_2 = c'_{j,j})
      If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m then S_1
      S_2
      If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m then S'_1
   enddo
enddo
do ( i'_1 = ig_j + 1, i'_1 = ig_{j+1} - 1 )
   do ( i'_2 = c_{j,j+1}, i'_2 = c'_{j,j+1})
      If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m then S_1
      S_2
      If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m then S'_1
   enddo
enddo
do ( i'_1 = ig_{j+1}, i'_1 = ig_{j+1} )
   do ( i'_2 = c_{j+1,j+1}, i'_2 = c'_{j+1,j+1})
      If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m then S_1
      S_2
      If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m then S'_1
   enddo
enddo
.
.
do ( i'_1 = ig_p, i'_1 = ig_p )
   do ( i'_2 = c_{p,p}, i'_2 = c'_{p,p})
      If h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m then S_1
      S_2
      If h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m then S'_1
   enddo
enddo
```

Figure 6.17: Code intermédiaire

6.3.4 Simplification d'un nid de boucles

Les conditions de la *proposition 24* (*page 109*) ne sont pas toujours vérifiées et la *proposition 25* (*page 112*) ne garantit pas que l'on puisse toujours extraire les codes S_1 et S'_1, car les droites définies par $h_m \ * \vec{j}' = b_m$ et $h_m \ * \vec{j}' = b_m$ peuvent apparaître soit sur la frontière inférieure, soit sur la frontière supérieure, soit perpendiculaires à l'axe i'_1. Afin de simplifier davantage les nids de boucles, on dissocie les cas précédents. Pour chaque nid de boucles du programme, on effectue les simplifications supplémentaires suivantes :

– La droite définie par $h_m \ * \vec{j}' = b_m$ est perpendiculaire à l'axe i'_1 (ce cas est possible uniquement pour le premier ou le dernier nid car la droite définie par $h_m \ * \vec{j}' = b_m$ est à la frontière du domaine d'itérations). On a $(h_{m,1} = 1) \wedge (h_{m,2} = 0) \wedge (b_m = ig_j : j = 1 \ ou \ j = p)$, on doit donc envisager deux possibilités :
   1. $b_m = ig_1 \Rightarrow$, on peut isoler le premier nid et on remplace "If $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ then S_1" par "S_1".
   2. $b_m = ig_p \Rightarrow$ on peut isoler le dernier nid et on remplace "If $h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m$ then S_1" par "S_1".

– La droite définie par $(h_m \ * \vec{j}' = b_m)$ apparaît sur la frontière inférieure c'est à dire $(h_{m,1} = a_j) \wedge (h_{m,2} = -d_j) \wedge (b_m = b_j)$. On doit envisager deux possibilités :
   1. $h_{m,2} = -1$: d'après la *proposition 24* (*page 109*) on peut sortir le code S_1 avant la deuxième boucle ;
   2. $h_{m,2} \neq -1$: d'après la *proposition 24* (*page 109*) on ne peut pas sortir le code S_1 avant la deuxième boucle mais on peut sortir à sa place "$If \ mod(b_m - h_{m,1} * i'_1 \ , \ h_{m,2}) = 0 \ then \ S_1$" car le code S_1 est exécuté uniquement pour des valeurs entières de $i'_2 = \frac{b_m - h_{m,1} * i'_1}{h_{m,2}}$.

– La droite définie par $(h_m \ \vec{j}' = b_m)$ apparaît sur la frontière supérieure, c'est à dire $(h_{m,1} = a'_j) \wedge (h_{m,2} = d'_j) \wedge (b_m = b'_j)$, on doit envisager deux possibilités :
   1. $h_{m,2} = 1$: d'après la *proposition 24* (*page 109*) on ne peut pas sortir le code S_1 avant la deuxième boucle mais on peut sortir "$S_1 \ ; \ S_2$" après la deuxième boucle car le code S_1 s'exécute uniquement au cours de la dernière itération de i'_2 et qu'il s'exécute avant le code S_2.
   2. $h_{m,2} \neq 1$: pour la même raison que précédemment et parce que le code S_1 s'exécute uniquement pour des valeurs entières de $i'_2 = \frac{b_m - h_{m,1} * i'_1}{h_{m,2}}$, on peut extraire après la deuxième boucle "$If \ mod(b_m - h_{m,1} * i'_1 \ , \ h_{m,2}) = 0 \ then \ S_1 \ ; S_2$".

   De manière symétrique nous avons les mêmes types de conditions associées à la génération et la simplification des gardes sur S'_1 :

– La droite définie par $h'_m \ * \vec{j}' = b'_m$ est perpendiculaire à l'axe i'_1 (ce cas est possible uniquement pour le premier ou le dernier nid). On a $(h'_{m,1} = 1) \wedge (h'_{m,2} = 0) \wedge (b'_m = ig_j : j = 1 \ ou \ j = p)$, on doit envisager deux possibilités :
   1. $b'_m = ig_1 \Rightarrow$ on peut isoler le premier nid et on remplace "If $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ then S'_1" par S'_1.
   2. $b'_m = ig_p \Rightarrow$ on peut isoler le dernier nid et on remplace "If $h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b'_m$ then S'_1" par S'_1.

– La droite définie par $h'_m \ * \vec{j}' = b'_m$ apparaît sur la frontière supérieure c'est à dire $(h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j) \wedge (b'_m = b'_j)$, on doit envisager deux possibilités
   1. $h'_{m,2} = 1$: d'après la *proposition 24* (*page 109*), on peut extraire le code S'_1 après la deuxième boucle.

2. $h'_{m,2} \neq 1$: d'après la *proposition 24* (*page 109*), on ne peut pas extraire le code S'_1 après la deuxième boucle mais on peut sortir à sa place $If\ mod(b'_m - h'_{m,1} * i'_1\ ,\ h'_{m,2}) = 0\ then\ S'_1$ car le code S'_1 est exécuté uniquement pour des valeurs entières de $i'_2 = \frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}}$.

– La droite définie par $h'_m * \vec{j}' = b'_m$ apparaît sur la frontière inférieure c'est à dire $(h'_{m,1} = a_j) \wedge (h'_{m,2} = -d_j) \wedge (b'_m = b_j)$, on doit envisager deux possibilités :

1. $h'_{m,2} = -1$: d'après la *proposition 24* (*page 109*), on ne peut pas extraire le code S'_1 après la deuxième boucle mais on peut sortir "S_2 ; S'_1" avant la deuxième boucle, car S'_1 s'exécute au cours de la première itération de i'_2 et qu'il s'exécute aussi après le code S_2.

2. $h'_{m,2} \neq -1$: pour la même raison que précédemment, et parce que le code S'_1 s'exécute que pour des valeurs entières de $i'_2 = \frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}}$, on extrait avant la deuxième boucle "$If\ mod(b'_m - h'_{m,1} * i'_1\ ,\ h'_{m,2})\ then\ S_2\ ;\ S'_1$".

6.3.5 Calcul de la borne inférieure et supérieure de la boucle la plus interne

On remarque que lorsqu'on extrait le code S_1 après (resp S'_1 avant) la deuxième boucle, on doit extraire avec lui le code S_2 ; par conséquent la borne supérieure (resp la borne inférieure) de la boucle la plus interne doit changer.

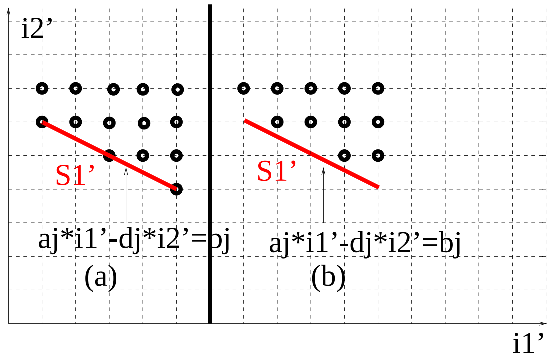


Figure 6.18: Avant et après l'extraction de S'_1

Si le demi-espace de la *figure 6.18*(a) est caractérisé par $(a_j * i'_1 - d_j * i'_2 \leq b_j) \Leftrightarrow (i'_2 \geq \frac{a_j * i_1 - b_j + d_j - 1}{d_j})$ alors le demi-espace de la*figure 6.18*(b) qui est obtenu à partir du demi-espace de la *figure 6.18*(a) en éliminant les points entiers sur le droite définie par $(a_j * i'_1 - d_j * i_2 = b_j)$, sera caractérisé par $(i'_2 \geq \frac{a_j * i'_1 - b_j}{d_j} + 1)$.

De même, si le demi-espace de la *figure 6.19*(a) est caractérisé par $(a'_i * i'_1 + d'_j * i'_2 \leq b'_j) \Leftrightarrow (i'_2 \leq \frac{-a'_j * i'_1 + b'_j}{d'_j})$, alors le demi-espace de la *figure 6.19*(b) qui est obtenu à partir du demi-espace de la *figure 6.19*(a) en éliminant les points entiers de la droite définie par $(a'_j * i'_1 + d'_j * i_2 = b'_j)$, sera caractérisé par $(i'_2 \leq \frac{-a'_j * i'_1 + b'_j}{d'_j} - \frac{1}{d_j})$.

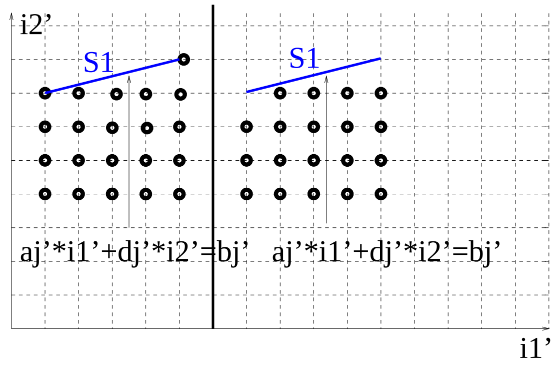


Figure 6.19: Avant et après l'extraction de S_1

6.4 Algorithme général

L'algorithme de génération de code se déroule en 4 étapes. Dans la première étape on découpe le nid transformé en une séquence de nids de manière à avoir une seule contrainte inférieure et une seule contrainte supérieure dans la boucle la plus interne de chaque nid. L'étape 2 traite le cas où les droites définies par $h_m * \vec{j}' = b_m$ et $h'_m * \vec{j}' = b'_m$ sont perpendiculaires à l'axe i'_1. Dans l'étape 3 on extrait les codes adéquats à l'extérieur de la boucle la plus interne et on calcule les bornes de la boucle la plus interne dans chaque nid. Enfin, dans la dernière étape on regroupe le code de certains sommets générateurs avec le code de l'intervalle gauche ou droite qui leur sont adjacents quand ceci est possible.

6.4.1 Etape 1

La première étape consiste à décomposer le nid de boucles transformé en plusieurs sous nids, en utilisant la projection des sommets générateurs du polytope transformé sur l'axe i'_1. Dans chaque nid on place avant et après la boucle la plus interne les codes $code_{i,j}$ et $code'_{i,j}$ qui seront calculés dans l'étape 3. Comme les droites définies par $h_m * \vec{j}' = b_m$ et $h'_m * \vec{j}' = b'_m$ peuvent être perpendiculaires à l'axe i'_1, alors dans le premier nid et le dernier nid on insère dans la boucle la plus interne des α_i et β_i qui seront calculés dans l'étape 2. Dans cette étape on génère le code en *figure 6.20*.

```
do ( $i'_1 = ig_1$, $i'_1 = ig_1$ )
    $code_{1,1}$
    do ( $i'_2 = c_{1,1}$, $i'_2 = c'_{1,1}$)
       $\alpha_1$
       $S_2$
       $\beta_1$
   enddo
    $code'_{1,1}$
enddo
.
.
do ( $i'_1 = ig_j$, $i'_1 = ig_j$ )
    $code_{j,j}$
    do ( $i'_2 = c_{j,j}$, $i'_2 = c'_{j,j}$)
       $S_2$
    enddo
    $codee'_{j,j}$
enddo
do ( $i'_1 = ig_j + 1$, $i'_1 = ig_{j+1} - 1$ )
    $code_{j,j+1}$
    do ( $i'_2 = c_{j,j+1}$, $i'_2 = c'_{j,j+1}$)
       $S_2$
   enddo
    $code'_{j,j+1}$
enddo
do ( $i'_1 = ig_{j+1}$, $i'_1 = ig_{j+1}$ )
    $code_{j+1,j+1}$
    do ( $i'_2 = c_{j+1,j+1}$, $i'_2 = c'_{j+1,j+1}$)
       $S_2$
   enddo
    $code'_{j+1,j+1}$
enddo
.
.
do ( $i'_1 = ig_p$, $i'_1 = ig_p$ )
    $code_{p,p}$
    do ( $i'_2 = c_{p,p}$, $i'_2 = c'_{p,p}$)
       $\alpha_p$
       $S_2$
       $\beta_p$
    enddo
    $code'_{p,p}$
enddo
```

Figure 6.20: Code intermédiaire

6.4.2 Etape 2

Comme on l'a mentionné précédemment, dans cette étape on calcule les $\alpha_i (i = 1\ ou\ p)$ et les $\beta_i (i = 1\ ou\ p)$ suivant les positions des droites définies par $h_m * \vec{j}' = b_m$ et $h'_m * \vec{j}' = b'_m$.

Si la droite définie par $h_m * \vec{j}' = b_m$ est perpendiculaire à l'axe i'_1 alors on doit envisager deux cas :

1. La droite définie par $h_m * \vec{j}' = b_m$ est dans la frontière gauche alors $\alpha_1 = S_1$.
2. La droite définie par $h_m * \vec{j}' = b_m$ est dans la frontière droite alors $\alpha_p = S_1$.

De même pour la droite définie par $h'_m * \vec{j}' = b'_m$ si elle est perpendiculaire à l'axe i'_1 alors on doit envisager deux possibilités :

1. La droite définie par $h'_m * \vec{j}' = b'_m$ est dans la frontière gauche alors $\beta_1 = S_1$.
2. La droite définie par $h'_m * \vec{j}' = b'_m$ est dans la frontière droite alors $\beta_p = S_1$.

Par contre si ces deux droites ne sont pas perpendiculaires à l'axe i'_1 alors les α_i et β_i seront éliminées dans la boucle la plus interne.

Par conséquent, les α_i et les β_i sont calculés de la manière suivante :

$$\alpha_1 = \begin{cases} \text{“ } S1 \text{ ''} & si\ (h_{m,0} = 1) \wedge (h_{m,1} = 0) \wedge (b_m = ig_1) \\ \text{“ ; ''} & sinon \end{cases}$$

$$\alpha_p = \begin{cases} \text{“ } S1 \text{ ''} & si\ (h_{m,0} = 1) \wedge (h_{m,1} = 0) \wedge (b_m = ig_p) \\ \text{“ ; ''} & sinon \end{cases}$$

$$\beta_1 = \begin{cases} \text{“ } S1' \text{ ''} & si\ (h'_{m,0} = 1) \wedge (h'_{m,1} = 0) \wedge (b'_m = ig_1) \\ \text{“ ; ''} & sinon \end{cases}$$

$$\beta_p = \begin{cases} \text{“ } S1' \text{ “} & si\ (h'_{m,0} = 1) \wedge (h'_{m,1} = 0) \wedge (b'_m = ig_p) \\ \text{“ ; ''} & sinon \end{cases}$$

6.4.3 Etape 3

Dans cette étape on doit d'abord localiser, s'il y a lieu, un sommet g_s qui satisfait la *condition*1 ou la *condition*2 car on effectue un traitement particulier sur le nid de boucles correspondant en utilisant la procédure *Traiterspeciale*. Les autres nids sont traités par la procédure *Traiter*. On peut schématiser cette étape par l'algorithme en *figure 6.21*.

$condition_1()$ et $condition_2()$ sont détaillées dans la *section 6.3.2*.

La procédure *Traiter* permet :

1. D'extraire les codes S_1 et S'_1 de la boucle la plus interne suivant les positions des droites définies par $h_m * \vec{j}' = b_m$ et $h'_m * \vec{j}' = b'_m$.
2. De calculer la contrainte inférieure et supérieure de la boucle la plus interne dans chaque nid, sachant que le choix de la contrainte inférieure dans C^- et de la contrainte supérieure dans C^+ qu'on affecte à un nid donné est présenté en *section 6.3.3*.

Le fonctionnement de la procédure *Traiterspeciale* est similaire à celui de la procédure *Traiter*, sauf que le code extrait avant ou après la deuxième boucle est différent.

```
soit nbr le nombre de nids générés.
si ∃ un sommet générateur g_s qui satisfait condition_1 ou condition_2 alors
   soit nid_k le nid corréspondant au sommet g_s
   do i=1, k-1
      traiter (nid_i)
   enddo
   traiterspecial(nid_k)
   do i=k+1, nbr
      traiter (nid_i)
   enddo
sinon
   do i=1, nbr
      traiter (nid_i)
   enddo
finsi
```

Figure 6.21: Algorithme de l'étape 3

Traiterspeciale

Cette procédure traite le nid de boucles qui correspond au sommet générateur qui satisfait la *condition 1* ou la *condition 2* :

$$code_{s,s} = \begin{cases} \text{“ } S_1 \text{ ; } S_2 \text{ ; } S_1' \text{ ''} & si\ condition_1(g_s) \\ \\ \text{“ ; ''} & sinon \end{cases}$$

$$code'_{s,s} = \begin{cases} \text{“ } S_1 \text{ ; } S_2 \text{ ; } S_1' \text{ ''} & si\ condition_2(g_s) \\ \\ \text{“ ; ''} & sinon \end{cases}$$

$$c_{s,s} = \begin{cases} \frac{a_j i_1 - b_j}{d_j} + 1 & si\ condition_1(g_s) \\ \\ \frac{a_j i_1 - b_j + d_j - 1}{d_j} & sinon \end{cases}$$

$$c'_{s,s} = \begin{cases} \frac{-a'_j i'_1 + b'_j}{d'_j} - \frac{1}{d_j} & si\ condition_2(g_s) \\ \\ \frac{-a'_j i'_1 + b'_j}{d'_j} & sinon \end{cases}$$

Procédure Traiter

Cette procédure calcule $code_{p,q}$, $garde'_{p,q}$, $c_{p,q}$ et $c'_{p,q}$ de la manière suivante :

$$code_{p,q} = \begin{cases} \text{“}S_1\text{''} & si\ (h_{m,1} = a_j)\wedge \\ & \quad (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \\ \text{“}If\ entier(\frac{b_m - h_{m,1} * i'_1}{h_{m,2}})\ then\ S_1\text{''} & si\ (h_{m,1} = a_j)\wedge \\ & \quad (h_{m,2} = -d_j \neq -1) \wedge (b_m = b_j) \\ \text{“}\ S_2\ ;\ S'_1\text{''} & si\ (h'_{m,1} = a_j)\wedge \\ & \quad (h'_{m,2} = -d_j = -1) \wedge (b'_m = b_j) \\ \text{“}\ If\ entier(\frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}})\ then\ S_2\ ;\ S'_1\text{''} & si\ (h'_{m,1} = a_j)\wedge \\ & \quad (h'_{m,2} = -d_j \neq -1) \wedge (b'_m = b_j) \\ \text{“}\ ;\ \text{''} & sinon \end{cases}$$

$$code'_{p,q} = \begin{cases} \text{“}\ S_1\ ;\ S_2\text{''} & si\ (h_{m,1} = a'_j)\wedge \\ & \quad (h_{m,2} = d'_j = 1) \wedge (b_m = b'_j) \\ \text{“}\ If\ entier(\frac{b_m - h_{m,1} i'_1}{h_{m,2}})\ then\ S_1\ ;\ S_2\text{''} & si\ (h_{m,1} = a'_j)\wedge \\ & \quad (h_{m,2} = d'_j \neq 1) \wedge (b_m = b'_j) \\ \text{“}\ S'_1\text{''} & si\ (h'_{m,1} = a'_j)\wedge \\ & \quad (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \\ \text{“}\ If\ entier(\frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}})\ then\ S'_1\text{''} & si\ (h'_{m,1} = a'_j)\wedge \\ & \quad (h'_{m,2} = d'_j \neq 1) \wedge (b'_m = b'_j) \\ \text{“}\ ;\ \text{“} & sinon \end{cases}$$

$$c_{p,q} = \begin{cases} \frac{a_j * i_1 - b_j}{d_j} + 1 & si\ (h'_{m,1} = a_j) \wedge (h'_{m,2} = -d_j) \wedge (b'_m = b_j) \\ \frac{a_j * i_1 - b_j + d_j - 1}{d_j} & sinon \end{cases}$$

$$c'_{p,q} = \begin{cases} \frac{-a'_j * i'_1 + b'_j}{d'_j} - \frac{1}{d_j} & si\ (h_{m,1} = a'_j) \wedge (h_{m,2} = d'_j) \wedge (b_m = b'_j) \\ \frac{-a'_j i'_1 + b'_j}{d'_j} & sinon \end{cases}$$

6.4.4 Etape 4

Pour tous les sommets générateurs qui ne sont pas dans l'un des cas de la *section 6.3.2*, on regroupe leur code avec le code de l'intervalle gauche ou droite qui leur sont adjacents.

6.5 Travaux antérieurs

Comme on l'a mentionné dans l'introduction il y a peu de travaux qui sont consacrés aux transformations directes de nids de boucles mal imbriquées.

Abu-Sufah [1] a proposé une technique de transformation d'un nid mal imbriqué en un nid parfaitement imbriqué en faisant entrer tous les codes à l'intérieur de la boucle la plus interne et en rajoutant des gardes sur ces codes, mais il ne s'est pas intéressé à la deuxième partie du problème qui est l'élimination des gardes dans le programme transformé.

Wolfe [80] a proposé d'utiliser la distribution de boucles afin de créer plusieurs nids de boucles parfaitement imbriquées, mais cette technique n'est pas utilisable en présence de dépendances cycliques.

Wolf [75] a proposé d'utiliser la même technique qu'Abu-Safuh et il a donné quelques heuristiques pour éliminer les gardes dans le programme transformé. Cependant, ces heuristiques ne permettent pas d'éliminer toutes les gardes.

Par contre Xue [83] s'est intéressé à traiter les deux parties du problème pour une certaine classe de nids de boucles mal imbriquées. Dans la première partie du problème il a utilisé la même technique qu'Abu-Safah pour transformer un nid de boucles mal imbriquées en un nid parfaitement imbriqué. Il a donné une condition d'application de cette technique mais il n'a pas traité les cas où cette condition n'est pas vérifiée. Pour la deuxième partie du problème, il a proposé une technique d'élimination des gardes qui est différente de la nôtre. Son principe consiste à découper le domaine d'itérations avant l'application de la transformation unimodulaire. Le code généré par cette technique est différent de celui généré par notre technique, qui de plus ne restreint pas les entrées.

On peut citer d'autres travaux qui sont consacrés à la génération du code mais qui ne se sont pas intéressé à ce problème en particulier. Parmi ces travaux on peut citer Chamski [21], Boulet [13], Quilleré [63] et Omega [47, 48].

Chamski [21] a proposé d'utiliser le découpage de domaines pour éliminer le calcul de maxima et de minima dans les bornes de boucles. Le travail de Boulet [13] génère aussi un code qui scanne les polyèdres. Le code généré est différent du nôtre car il ne contient pas des boucles "DO" et pas donc des problèmes spécifiques aux optimiseurs.

L'Omega [47, 48] et Quileré [63] génèrent des codes qui scannent un ensemble de polyèdres convexes. Le code généré par ces deux derniers est presque similaire à notre code, mais ils se basent sur des algorithmes différents du nôtre.

L'Omega calculateur [47, 48] propose une méthode non récursive, en prenant en entrée un code parfaitement imbriqué qui scanne une union de domaines, et il essaye d'éliminer les gardes. L'élimination d'une garde conduit à générer deux boucles telles que dans l'une la condition de la garde est toujours satisfaite et dans l'autre la condition ne l'est jamais. Dans certains cas il génère un code faux : si on prend l'exemple de la *figure 6.24*, il sort les codes gardés, à l'extérieur de la boucle la plus interne sans les gardes "if entier $(\frac{i'}{6})''$ et "if entier $(\frac{i'-6}{6})$".

Par contre, Quileré [63] décompose récursivement une union de polyèdres en un nid de boucles, en commençant par la boucle la plus externe et en allant jusqu'à la boucle la plus interne. A chaque étape de la récursion il génère un ensemble de boucles supplémentaires. Afin d'ordonner ces boucles il a défini un ordre partiel sur les polyèdres. Le code généré par celui-ci est presque similaire au nôtre, à l'exception de certains cas où il isole inutilement certains sous-ensembles du domaine d'itération sans que cela soit nécessaire et au prix d'une augmentation de la taille du code, comme on peut le voir sur les *figures 6.22* et *6.27*.

Notre travail est donc très proche de celui de Quilleré [63] et de l'Omega [47, 48] calculateur. Comme nous traitons un cas particulier, notre algorithme est plus rapide et plus facile à implémenter. Les *figures 6.22*, *6.23* et *6.24* montrent respectivement les codes générés par l'algorithme de Quilleré [63], par notre algorithme et par l'Omega calculateur [47, 48] pour l'exemple du produit matrice vecteur, après l'application de la transformation unimodulaire qui est décrite en section 6.1.

On remarque sur les *figures 6.22* et *6.24* que l'algorithme de Quilleré [63] isole les codes correspondant à i = 0 et i=30 sans que cela soit nécessaire et que l'Omega calculateur [47, 48] génère un code faux parce qu'il omet des gardes nécessaires pour les instructions Y(j)=S et S=0. En effet ces dernières s'exécutent uniquement pour des valeurs de i qui sont des multiples de 6. Par contre les gardes $if\ (\ i\ \%\ 6\ ==\ 0)$ et $if\ (\ (i-6\ )\%\ 6\ ==\ 0)$

respectivement dans le deuxième et l'avant dernier nid du code généré par l'algorithme de Quilleré sont inutiles.

Les *figures 6.26* et 6.27 montrent respectivement les codes générés par notre algorithme et par l'algorithme de Quilleré [63] pour l'exemple de la *figure 6.25* après l'élimination des gardes. On voit que l'algorithme de Quilleré [63] isole inutilement certains sous-ensembles du domaine d'itérations sans que cela soit nécessaire.

6.6 Conclusion

L'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées a été traitée par peu d'auteurs. Nous avons traité les deux problèmes en détail : (1) la mise sous forme parfaitement imbriquée d'un nid de boucles et (2) l'élimination de gardes dans le nid transformé. Pour la première partie, nous avons donné la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid parfaitement imbriqué. Si cette condition n'est pas satisfaite, ou bien (1) la boucle interne est vide et nous éliminons entièrement le corps de la boucle interne, ou (2) nous décomposons le nid de boucles en une séquence de deux nids de boucles. Dans un nid, nous faisons rentrer toutes les instructions dans la boucle la plus interne et dans l'autre nous éliminons le corps de la boucle interne.

Pour la seconde partie du problème, nous avons proposé un algorithme qui permet d'éliminer les gardes, en décomposant le nid transformé en une séquence de nids de boucles. Notre algorithme minimise le nombre de nids de boucles nécessaires à l'élimination de toutes les gardes internes. Il se peut que des gardes subsistent dans le corps de la boucle externe. Elles pourraient aussi être éliminées par un déroulage de cette boucle, qui aurait l'inconvénient d'augmenter la taille du code. Ces contributions sont comparées à l'état de l'art dans la *Section 6.5.*

Vu la difficulté de la deuxième partie de ce problème, nous avons limité notre étude aux nids de boucles à deux dimensions, non paramétriques, parce que le nombre de formes que peut prendre l'algorithme décomposé est déjà très élevé et l'expansion de code non négligeable dans les cas les plus défavorables. Dans de prochains travaux, nous espérons pouvoir généraliser cette étude au cas général, paramétrique et en dimension quelconque, mais la taille du code produit risque d'exploser. Il sera sans doute nécessaire de se restreindre aux boucles les plus internes pour lesquelles l'élimination de garde permet d'appliquer un algorithme de pipeline logiciel et produit de toute façon la partie essentielle de l'optimisation puisque ce sont les tests les plus fréquemment exécutés.

```
{ i = 0;
   { j = 0;
      S=0; S=S+A(0,0)*X(0);
   }
}
for (i = 1; i <= 5; i++) {
   for (j = 0; j <= floord(i-1,6); j++) {
      S=S+A(j,i-6*j)*X(i-6*j);
   }
   if ((i)%6 == 0) { j=i/6;
      S=0; S=S+A(j,0)*X(0);
   }
}
for (i = 6; i <= 24; i++) {
   if ((i-6)%6 == 0) { j=(i-6)/6;
      S=S+A(j,6)*X(6); Y(j)=S;
   }
   for (j = ceild(i-5,6); j <= floord(i-1,6); j++)
      S=S+A(j,i-6*j)*X(i-6*j);
   }
   if ((i)%6 == 0) { j= i/6
      S=0; S=S+A(j,0)*X(0);
   }
}
for (i = 25; i <= 29; i++) {
   if ((i-6)%6 == 0) { j= (i-6)/6;
      S=S+A(j,6)*X(6); Y(j)=S;
   }
   for (j = ceild(i-5,6); j <= 4; j++) {
      S=S+A(j,i-6*j)*X(i-6*j);
   }
}
{ i = 30;
   { j = 4;
      S=S+A(4,6)*X(6); Y(4)=S;
   }
}
```

Figure 6.22: Produit Matrice /Vecteur : Code généré par l'algorithme de Quilleré

```
do ( i' = 0, i' = 5 )
   do ( j' = 0 , j' = i'/6 − 1/6)
     S = S + A(j', i' − 6 * j') * X(i' − 6 * j')
   enddo
   if (entier i'/6) then {  S=0 ; S = S + A(i'/6, 0) * X(0) }
 enddo
do ( i' = 6, i' = 24 )
   if (entier (i'−6)/6) then { S = S + A((i'−6)/6, 6) * X(6) ; Y(((i'−6)/6) = S}
   do ( j' = i/6 , j' = i'/6 − 1/6)
     S = S + A(j', i' − 6 * j') * X(i' − 6 * j')
   enddo
   if (entier i'/6) then { S=0 ; S = S + A(i'/6, 0) * X(0) }
   enddo
do ( i' = 25, i' = 30 )
   if (entier (i'−6)/6) then { S = S + A((i'−6)/6, 6) * X(6);  Y((i'−6)/6) = S }
   do ( j' = i/6 , j' = 4)
     S = S + A(j', i' − 6 * j') * X(i' − 6 * j')
   enddo
enddo
```

Figure 6.23: Produit Matrice/Vecteur : Code généré par notre algorithme

```
for  (i = 0 ; i <= 30 ; i++) {
   j=intDiv((i+5),6) ;
   if (i>= 6) {
      S=S+A(j,6)*X(6) ;
   }
   if ( i >= 6) {
      Y(j)=S ;
   }
   for (j=intDiv(i-5+5,6) ;j<=intDiv(i-1,6) ;j++) {
      S=S+A(j,i-6*j)*X(i-6*j)
   }
   j=intDiv((i+5),6) ;
   if (i<= 24) {
      S=0 ;
   }
   if I <= 24) {
      S=S+A(j,0)*X(0)
   }
}
```

Figure 6.24: Produit Matrice/Vecteur : Code généré par l'Omega calculateur

```
for (i = 0 ; i <= 5 ; i++) {
    for (j = i ; j <= 5 ; j++) {
        if (i-j==0) S1 ;
        S2 ;
        if (j-5==0) S3 ;
    }
}
```

Figure 6.25: Exemple générique : Code avant l'élimination des gardes

```
for (i = 0 ; i <= 5 ; i++) {
    S1 ;
    for (j = i ; j <= 5 ; j++) {
        S2 ;
    }
    S3 ;
}
```

Figure 6.26: Exemple générique : Code généré par notre algorithme

```
for (i = 0 ; i <= 3 ; i++) {
    { j = i ;
        S1 ; S2 ;
    }
    for (j = i+1 ; j <= 4 ; j++) {
        S2 ;
    }
    { j = 5 ;
        S2 ; S3 ;
    }
}
{ i = 4 ;
    { j = 4 ;
        S1 ; S2 ;
    }
    { j = 5 ;
        S2 ; S3 ;
    }
}
{ i = 5 ;
    { j = 5 ;
        S1 ; S2 ; S3 ;
    }
}
```

Figure 6.27: Exemple générique : Code généré par l'algorithme de Quilleré

Chapitre 7

Implantation et expérimentations

Nous rappelons d'abord les algorithmes qui ont été implantés dans *PIPS*, avec chaque fois un petit exemple d'utilisation. Nous expliquons ensuite quelles grandeurs nous avons mesurées et pourquoi. L'ensemble des transformations proposées est alors appliqué à un cas d'école puis à une application, la détection de cavités.

7.1 Implantation

À l'exception de la fusion de boucles avec pavage à deux niveaux et avec le remplacement de chaque tableau par plusieurs tampons et registres, toutes les transformations détaillées dans les chapitres précédents sont implantées dans *Pips* [64] (Compilateur de Recherche du *CRI* à l'École des mines de Paris). Ces transformations, nous pouvons les diviser en deux grands groupes :

1. Les transformations visant à réduire la consommation de l'énergie électrique dans le cas des applications de traitement du signal. Comme nous l'avons mentionné dans les chapitres précédents, les codes de ces applications sont des séquences de nids de boucles de profondeur quelconque. Ces transformations sont :
   (a) la fusion d'une séquence de nids de boucles ;
   (b) la fusion d'une séquence de nids de boucles avec le remplacement de chaque tableau A_k par un tampon séquentiel et circulaire ;
   (c) la fusion avec pavage à un seul niveau d'une séquence de nids de boucles ;
   (d) la fusion avec pavage à un seul niveau d'une séquence de nids de boucles et avec le remplacement de chaque tableau A_k par plusieurs tampons ;
   (e) la fusion avec pavage à deux niveaux d'une séquence de nids de boucles ;
   (f) la fusion avec pavage à deux niveaux d'une séquence de nids de boucles et avec le remplacement de chaque tableau A_k par plusieurs tampons et registres. *Cette transformation sera implantée dans les travaux futurs.*
2. Une transformation visant à éliminer les gardes dans les transformées unimodulaires de nid de boucles mal imbriquées. Les codes en entrée de cette transformation sont des nids de boucles mal imbriquées de profondeur 2.

7.1.1 Fusion d'une séquence de nids de boucles

La fusion d'une séquence de nids de boucles (voir *chapitre 3*) est implantée suivant l'algorithme général donné en *figure 7.1*.

Début
1) identifier le stencil associé à chaque $nid\ k$.
2) vérifier les hypothèses données dans la *section 3.1* (*page 11*) sur les différents domaines D_k.
3) calculer le vecteur de décalage $\vec{h}_k$ de chaque $nid\ k$ en utilisant le *théorème 1* (*page 16*).
4) calculer la translation du domaine D_k par le vecteur $\vec{h}_k$.
5) calculer l'enveloppe convexe du domaine d'itération du nid fusionné.
6) calculer la garde de l'instruction S_k du $nid\ k$.
7) fusionner tous les nids en un seul nid (*figure 3.8* en page 15).

Fin

Figure 7.1: Algorithme général de la fusion d'une séquence de nids de boucles

Les codes en *figures* 7.25 (*page 147*) et 7.26 (*page 147*) montrent respectivement un exemple de code constitué de deux nids de boucles et son code fusionné que nous obtenons par notre implantation dans *Pips*. Le système qui permet d'appliquer des transformations est appelé *tpips* : celui-ci reçoit des commandes permettant d'effectuer diverses transformations. La *figure 7.2* montre comment la fusion est appliquée sur le code fortran *exemple.f* via *tpips*.

```
$ tpips
create exemple exemple.f
setproperty PRETTYPRINT_ALL_DECLARATIONS TRUE
display PRINTED_FILE[TRAITE]
apply FUSION[TRAITE]
apply CLEAN_DECLARATIONS[TRAITE]
apply UNSPLIT
close
$ f77 exemple.database/Src/exemple.f
```

Figure 7.2: Effectuer une transformation de fusion via tpips, sur le code fortran exemple.f

7.1.2 Fusion d'une séquence de nids de boucles avec l'allocation des tampons

Toutes les étapes de l'implantation de la fusion sont utilisées dans l'implantation de la fusion avec l'allocation des tampons. Nous avons rajouté dans cette implantation : (1) le calcul du volume mémoire de données vivantes de chaque tableau A_k, (2) le calcul de la fonction d'accès de chaque tampon B_k associé au tableau A_k et (3) le remplacement de chaque tableau A_k par le tampon B_k. L'algorithme général de cette implantation est donné en *figure 7.3* (*page 131*).

Le code en *figure 7.27* (*page 148*) donne la fusion du code en *figure 7.25* (*page 147*) avec le remplacement du tableau B par le tampon B_1. La *figure 7.4* (*page 131*) montre comment la fusion avec allocation des tampons est appliquée sur le code fortran *exemple.f* via *tpips*.

Début

1) identifier le stencil associé à chaque $nid\ k$.
2) vérifier les hypothèses données dans la *section 3.1* (*page 11*) sur les différents domaines D_k.
3) calculer le vecteur de décalage $\vec{h}_k$ de chaque $nid\ k$ en utilisant le *théorème 1* (*page 16*).
4) calculer la translation du domaine D_k par le vecteur $\vec{h}_k$.
5) calculer l'enveloppe convexe du domaine d'itération du nid fusionné.
6) calculer la garde de l'instruction S_k du $nid\ k$.
7) fusionner tous les nids en un seul nid.
8) calculer le volume mémoire de données vivantes de chaque tableau A_k en utilisant le *théorème 2* (*page 20*).
9) calculer la fonction d'accès du tampon B_k associé au tableau A_k en utilisant la *proposition 5* (*page 23*).
10) remplacer le tableau A_k par le tampon B_k.

Fin

Figure 7.3: Algorithme général de la fusion d'une séquence de nids de boucles avec l'allocation des tampons

```
$ tpips
create exemple exemple.f
setproperty PRETTYPRINT_ALL_DECLARATIONS TRUE
display PRINTED_FILE[TRAITE]
apply FUSION_AND_TAMPON[TRAITE]
apply CLEAN_DECLARATIONS[TRAITE]
apply UNSPLIT
close
$ f77 exemple.database/Src.exemple.f
```

Figure 7.4: Effectuer une transformation de fusion avec allocation des tampons via tpips, sur le code fortran exemple.f

7.1.3 Fusion avec pavage à un seul niveau avec allocation des tampons

Les étapes les plus importantes de cette implantation sont données dans l'algorithme en *figure 7.5* (*page 132*) qui sont divisées en trois grandes parties : (1) le calcul de la translation des différents domaines d'itérations, (2) l'application de la fusion avec pavage et (3) l'allocation des tampons. Comme nous pouvons le remarquer dans cet algorithme, l'implantation de la fusion avec pavage à un seul niveau d'une séquence de nids de boucles est incluse dans cet algorithme. C'est pour cette raison, que nous ne l'avons pas décrite dans ce chapitre.

Le code en *figure 7.28* (*page 149*) donne la fusion avec pavage à un seul niveau du code en *figure 7.25* (*page 147*) avec le remplacement du tableau B par plusieurs tampons. La *figure 7.6* (*page 132*) montre comment la fusion avec pavage à un seul niveau et avec allocation des tampons est appliquée sur le code fortran *exemple.f* via *tpips*.

Début
1) identifier le stencil associé à chaque $nid\ k$.
2) vérifier les hypothèses données dans la *section 3.1* (*page 11*) sur les différents domaines D_k.
3) calculer le vecteur de décalage $\vec{h}_k$ de chaque $nid\ k$ en utilisant le *théorème 5* (*page 42*).
4) calculer la translation du domaine D_k par le vecteur $\vec{h}_k$.
5) calculer l'enveloppe convexe du domaine d'itération du nid tuilé et fusionné.
6) calculer la garde de chaque instruction S_k du $nid\ k$.
7) appliquer la matrice du pavage $\mathcal{A}$ sur le code après la translation des différents domaines.
8) appliquer la matrice $\mathcal{P}$ sur le code obtenu après l'application de la matrice $\mathcal{A}$.
9) remplacer chaque tableau tableau A_k par $2n + 1$ tampons.
Fin

Figure 7.5: Algorithme général de la fusion avec pavage à un seul niveau et avec le remplacement de chaque tableau par plusieurs tampons

```
$ tpips
create exemple exemple.f
setproperty PRETTYPRINT_ALL_DECLARATIONS TRUE
display PRINTED_FILE[TRAITE]
apply PAVAGE1_SEQUENCE[TRAITE]
A
P
apply CLEAN_DECLARATIONS[TRAITE]
apply UNSPLIT
close
$ f77 exemple.database/Src/exemple.f
```

Figure 7.6: Effectuer une transformation de fusion avec pavage à un seul niveau et avec allocation des tampons via tpips, sur le code fortran exemple.f

7.1.4 Fusion avec pavage à deux niveaux et avec l'allocation des tampons et des registres

Les grandes étapes de cette technique sont implantées dans *Pips* : (1) le pavage à deux niveaux, (2) le déroulage de boucles et (3) l'allocation des tampons. Il reste à implanter l'allocation des registres et à combiner toutes ces étapes. Le code en *figure 7.29* (*page 150*) donne la fusion avec pavage à deux niveaux du code en *figure 7.25* (*page 147*).

7.1.5 Élimination des gardes

Comme nous l'avons mentionné précédemment, les codes en entrée de cette technique sont des nids de boucles de profondeur 2. L'algorithme général de cette implantation est donné en *figure 7.7* (*page 133*).

Le code en *figure 7.30* (*page 150*) montre un exemple de code généré dans notre implantation après l'application d'une transformation unimodulaire et l'élimination des gardes dans le nid transformé. La *figure 7.8* (*page 133*) montre comment l'élimination des gardes est appliquée sur le code fortran *exemple.f* via *tpips*.

```
début
  1) transformer le nid de boucles mal imbriquées en un nid parfaitement imbriqué
     en utilisant la proposition 23 (page 105).
  2) appliquer la transformation unimodulaire sur le nid parfaitement imbriqué de profondeur 2.
  3) éliminer les gardes dans le nid transformé en décomposant celui ci en plusieurs sous-nids
     suivants les sommets générateurs.
Fin
```

Figure 7.7: Algorithme général de l'élimination des gardes dans les transformées unimodulaires de nids de boucles mal imbriquées

```
$ tpips
create exemple exemple.f
setproperty PRETTYPRINT_ALL_DECLARATIONS TRUE
display PRINTED_FILE[TRAITE]
apply GUARD_ELIMINATION[TRAITE]
P
apply CLEAN_DECLARATIONS[TRAITE]
apply UNSPLIT
close
$ f77 exemple.database/Src/exemple.f
```

Figure 7.8: Effectuer une élimination des gardes via *tpips*, sur le code fortran exemple.f

7.2 Mesures effectuées

Comme nous l'avons mentionné précédemment, l'objectif de cette thèse est de minimiser la consommation de l'énergie électrique dans le cas des applications de traitement du signal, qui est fortement dépendante des transferts des données entre les différents niveaux de la hiérarchie mémoire. Les mesures effectuées dans ce chapitre sont : (1) les défauts de cache externe L_2, (2) les temps d'exécution, (3) les tailles des codes et (4) les tailles mémoire occupées par les différents tableaux et tampons. Les mesures (2), (3) et (4) ne jouent pas un rôle important dans le modèle de consommation de l'énergie électrique par rapport aux défauts de cache L_2. Nous avons considéré uniquement les défauts de cache externe L_2 ; c'est le seul qui génère des transferts de données entre le processeur et l'extérieur, le cache L_1 étant intégré. Les différentes mesures citées précédemment sont effectuées sur trois machines cibles qui sont :

1. *Sun-Blade-1000*, à base d'un microprocesseur *UltraSPARC-III* dont la taille du cache externe est 8 M octets. Il est organisé par ligne de 512 octets mais le chargement de données dans ce cache se fait par sous-blocs de 64 octets. La fréquence d'horloge de cette machine est de 750 MHZ.
2. *Ultra 10*, à base d'un microprocesseur *UltraSPARC-IIi* avec un cache externe de 2 M octets, divisé en ligne de 64 octets. La fréquence d'horloge de cette machine est de 440 MHZ.
3. *Ultra 5*, à base d'un microprocesseur *UltraSPARC-IIi* avec un cache externe de 256 k octets, divisé en ligne de 64 octets. La fréquence d'horloge de cette machine est de 360 MHZ.

7.3 Cas d'école

Le code considéré dans cette section est donné en *figure 7.9* (*page 135*). Ce code est constitué de deux nids de boucles de profondeur 2. Le patron associé au premier nid est constitué de 5 références au tableau A_0 et le patron associé au deuxième nid est constitué de 5 références au tableau A_1. Nous avons mesuré les défauts de cache, les temps d'exécution des différentes transformations de ce code, les tailles des codes et les tailles mémoire.

7.3.1 Défauts de cache externe (L_2)

Comme nous l'avons mentionné précédemment, pour effectuer nos expériences, nous avons utilisé trois machines : *Sun-Blade-1000*, *Ultra 10* et *Ultra 5*. Nous avons utilisé les fonctions de comptage d'événements matériels, issues de la librairie *libcpc.h*, pour mesurer les défauts de cache externe engendrés par les différentes transformations du code d'école donné en *figure 7.9* :

1. code initial ;
2. code fusionné ;
3. code fusionné avec allocation des tampons ;
4. code fusionné avec pavage à un seul niveau et avec allocation des tampons ;
5. code fusionné avec pavage à deux niveaux.

Les *figures 7.10*, *7.11* et *7.12* donnent les défauts de cache externe causés par ces différents codes en fonction de la taille des tableaux respectivement sur *Sun-Blade-1000*, *Ultra 10* et *Ultra 5*. Tous les tableaux ont la même taille qui est égale à $N * N$ avec $1000 \leq N \leq 6080$ sur *Sun-Blade-1000* et *Ultra 10* et $1000 \leq N \leq 3530$ sur *Ultra 5*. Nous pouvons remarquer sur les machines *Sun-Blade-1000* et *Ultra 10* que :

1. le code fusionné avec pavage à un seul niveau et avec allocation des tampons donne le meilleur résultat (le minimum de défauts de cache).
2. le code fusionné avec allocation des tampons donne un résultat meilleur que le code fusionné avec pavage.
3. l'allocation de tampons dans le code fusionné et dans le code fusionné avec pavage permet de réduire de manière considérable le nombre de défauts de cache.
4. pratiquement toutes les mesures suivent asymptotiquement des droites bien définies qui correspondent aux résultats théoriques attendus. Le calcul des pentes de ces droites est détaillé ultérieurement dans le cas de l'exemple de détection de cavité et elles sont données par :
   (a) $\frac{16}{L}$ pour le code initial ;
   (b) $\frac{12}{L}$ pour le code fusionné ;
   (c) $\frac{12}{L}$ pour le code fusionné avec pavage ;
   (d) $\frac{8}{L}$ pour le code fusionné avec allocation des tampons ;
   (e) $\frac{8}{L}$ pour le code fusionné avec pavage et avec allocation des tampons.

   L représente la taille des blocs chargés dans le cache externe. Dans toutes les machines cibles utilisées L vaut 64.

Cependant sur la machine *Ultra 5*, les mesures ne suivent pas les droites théoriques définies précédemment et c'est le code fusionné avec pavage qui donne le meilleur résultat. Ces résultats s'expliquent par la petite taille du cache externe de cette machine.

do $(i = 5,\ N - 4)$

  do $(j = 5,\ N - 4)$

$$
\begin{aligned}
A_1(i,\ j) = {} & A_0(i-4,\ j) + A_0(i,\ j-4) \\
& + A_0(i,\ j) + A_1(i,\ j+4) \\
& + A_0(i+4,\ j)
\end{aligned}
$$

  enddo

enddo

do $(i = 9,\ N - 8)$

  do $(j = 9,\ N - 8)$

$$
\begin{aligned}
A_2(i,\ j) = {} & A_1(i-4,\ j) + A_1(i,\ j-4) \\
& + A_1(i,\ j) + A_1(i,\ j+4) \\
& + A_1(i+4,\ j)
\end{aligned}
$$

  enddo

enddo

Figure 7.9: Cas d'école

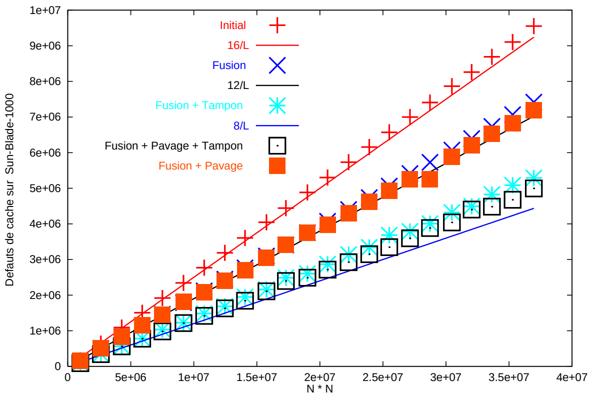


Figure 7.10: Défauts de cache externe (L2) pour les différentes transformations du code de l'exemple en *figure 7.9* sur *Sun-Blade-1000*

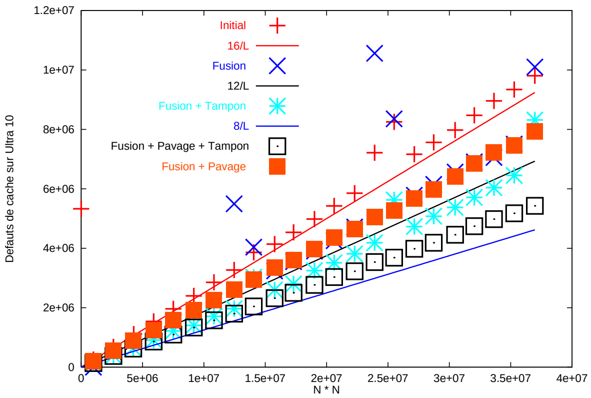


Figure 7.11: Défauts de cache externe (L2) pour les différentes transformations du code de l'exemple en *figure 7.9* sur *Ultra 10*

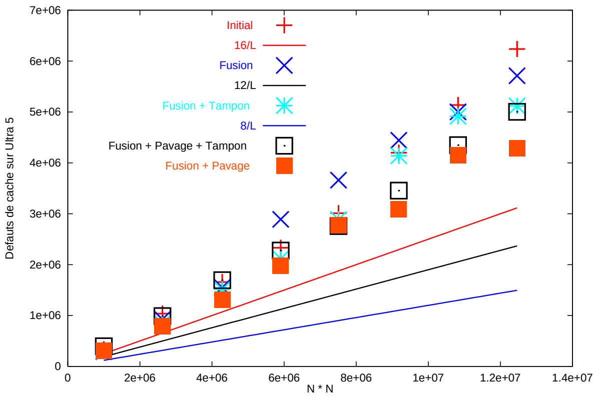


Figure 7.12: Défauts de cache externe (L2) pour les différentes transformations du code de l'exemple en *figure 7.9* sur *Ultra 5*

7.3.2 Temps d'exécution

Comme nous l'avons mentionné précédemment, l'objectif de cette thèse est l'augmentation de la localité des données. Nous avons néanmoins mesuré les temps d'exécution des différentes transformations du code en *figure 7.9* :

1. code initial ;
2. code fusionné ;
3. code fusionné avec allocation des tampons ;
4. code fusionné avec pavage à un seul niveau et avec allocation des tampons ;
5. code fusionné avec pavage à deux niveaux.

Les *figures 7.13*, *7.14* et *7.15* donnent les temps d'exécutions en fonction de la taille des tableaux respectivement sur *Sun-Blade-1000*, *Ultra 10* et *Ultra 5*.

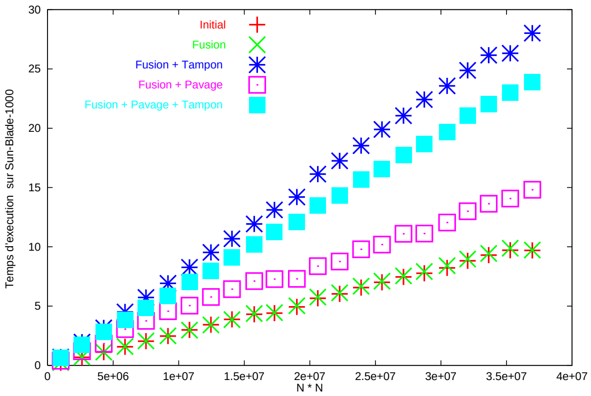


Figure 7.13: Temps d'exécution des différentes transformations du code de l'exemple en *figure 7.9* sur *Sun-Blade-1000*

Nous pouvons remarquer que :

1. Le code fusionné avec allocation des tampons augmente considérablement le temps d'exécution. Cette augmentation est prévisible, car les *modulos* introduits dans les

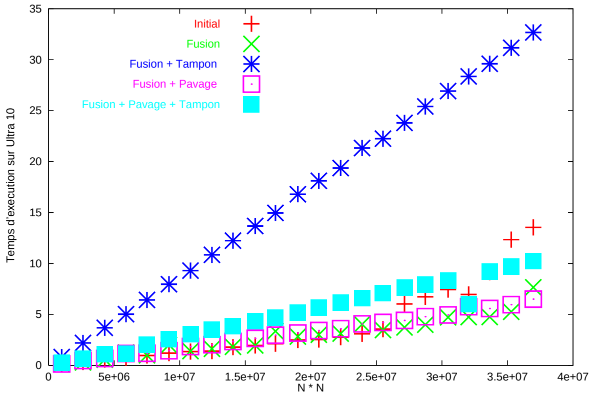


Figure 7.14: Temps d'exécution des différentes transformations du code de l'exemple en *figure 7.9* sur *Ultra 10*

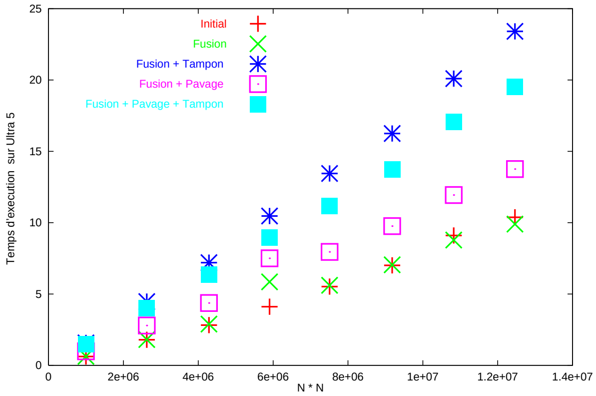


Figure 7.15: Temps d'exécution des différentes transformations du code de l'exemple en *figure 7.9* sur *Ultra 5*

fonctions d'accès des différents tampons consomment beaucoup de temps. Pour améliorer le temps d'exécution de ce code nous pouvons soit (1) prendre des puissances de 2 pour les tailles des tampons soit (2) éliminer les *modulos* par le déroulage de boucles. Dans l'exemple de détection de cavités, nous montrons que les tailles des tampons qui sont égales à des puissances de 2 permettent de réduire considérablement le temps d'exécution du code fusionné avec allocation des tampons.

2. Le temps d'exécution du code fusionné avec pavage et avec allocation des tampons est inférieur au temps d'exécution du code fusionné avec allocation des tampons. Mais il est supérieur aux temps d'exécution des autres codes. Cela est dû aux gardes que nous avons rajouté pour savoir dans quel tampon nous récupérons et nous sauvegardons les données du tableau correspondant. Nous pouvons réduire ces gardes en prévoyant statiquement la lecture et l'écriture dans les différents tampons.

7.3.3 Taille mémoire

Ici nous donnons les tailles mémoires en octet des différents tableaux et tampons utilisés dans les différentes transformations.

1) Code initial :

Dans le code initial nous avons 3 tableaux; chacun de ces tableaux a une taille de $4 * N^2$. Par conséquent, le volume mémoire de tous les tableaux manipulés dans ce code est :

$$12 * N^2$$

2) Code fusionné :

Le nombre de tableaux utilisés dans ce code est le même que celui utilisé dans le code initial. Le volume mémoire de tous les tableaux manipulés dans ce code est donc :

$$12 * N^2$$

3) Code fusionné avec allocation des tampons :

Dans ce code nous avons deux tableaux de taille $4 * N^2$ chacun et un tampon B_1 de taille $4 * (8N - 63)$. le volume mémoire total est donc :

$$8 * N^2 + 32 * N - 252$$

4) Code tuilé :

Le nombre de tableaux utilisés dans ce code est le même que celui utilisé dans le code initial. Le volume mémoire de tous les tableaux manipulés dans ce code est donc :

$$12 * N^2$$

5) Code fusionné avec pavage et avec allocation des tampons :

Dans ce code nous avons deux tableaux de taille $4 * N^2$ chacun et 3 tampons $B_{1,1}$, $B_{1,2}$ et $B_{1,3}$. Les tailles de ces tampons sont respectivement $64 * (N - 8)$, $64 * T_1$ et $36 * T_2$. Les coefficients T_1 et T_2 sont respectivement les tailles des tuiles sur les axes i et j. Le volume mémoire de tous les tableaux et tampons manipulés dans ce code est donc :

$$8 * N^2 + 64 * N + 64 * T_1 + 36 * T_2 - 512$$

7.3.4 Taille des codes

Le tableau suivant donne les tailles du code initial et de ses différentes transformations :

Les codes	Taille du code *Fortran*	Taille du code *Assembleur*
Initial	10	118
Fusion	8	161
Fusion + Tampon	10	252
Fusion + Pavage	27	295
Fusion + Pavage + Tampon	70	930

Nous pouvons remarquer que le code fusionné avec pavage et avec allocation des tampons possède la plus grande taille. Cela est toujours dû aux tests que nous avons ajoutés pour savoir dans quel tampon nous sauvegardons et nous récupérons la donnée du tableau correspondant. Nous pouvons réduire cette taille en prévoyant statiquement ces lectures et ces écritures.

7.4 Détection de cavités

Ici, nous donnons un exemple de code en *figure 7.31* (*page 151*) qui est une application réelle d'IMEC. Cette application récupère une image en entrée avec la fonction *Lireimage* sur laquelle nous effectuons cinq traitements différents qui correspondent aux cinq nids de la fonction *TRAITE*.

7.4.1 Défauts de cache externe

Nous avons mesuré les défauts de cache externe engendrés par les différentes transformations du code cavité donné en *figure 7.31* (*page 151*) :

1. code initial ;
2. code fusionné ;
3. code fusionné avec allocation des tampons ;
4. code fusionné avec pavage à un seul niveau et avec allocation des tampons ;
5. code fusionné avec pavage à deux niveaux.

Les *figures* 7.16, 7.17 et 7.18 donnent les défauts de cache externe causés par ces différents codes en fonction de la taille des tableaux respectivement sur *Sun-Blade-1000*, *Ultra 10* et *Ulra 5*. Tous les tableaux ont la même taille qui est égale à $N_1 * N_2$ avec $1280 \leq N_1 \leq 6400$ et $1200 \leq N_2 \leq 3000$ sur *Sun-Blade-1000* et $640 \leq N_1 \leq 1280$ et $200 \leq N_2 \leq 1200$ sur *Ultra 5* et *Ultra 10*.

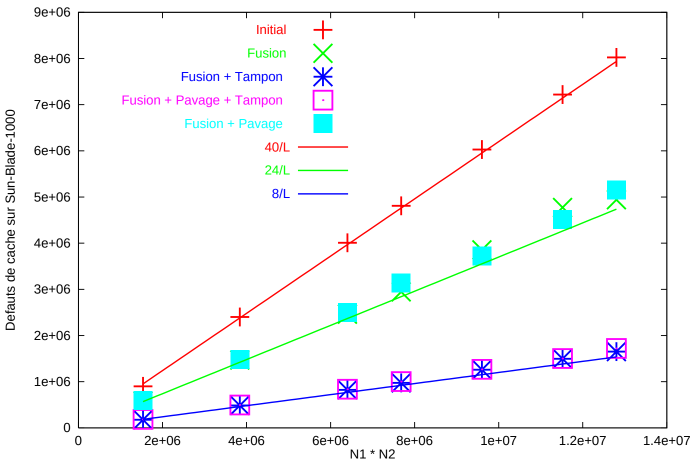


Figure 7.16: Défauts de cache externe (L2) sur *Sun-Blade-1000* pour les différentes transformations du code de l'exemple en *figure 7.31*

Nous pouvons remarquer que l'allocation des tampons dans le cas de la fusion et dans le cas de la fusion avec pavage permet de réduire d'une manière considérable le nombre de défauts de cache sur toutes les machines cibles précédentes. Sur *Sun-Blade-1000* et *Ultra 10*,

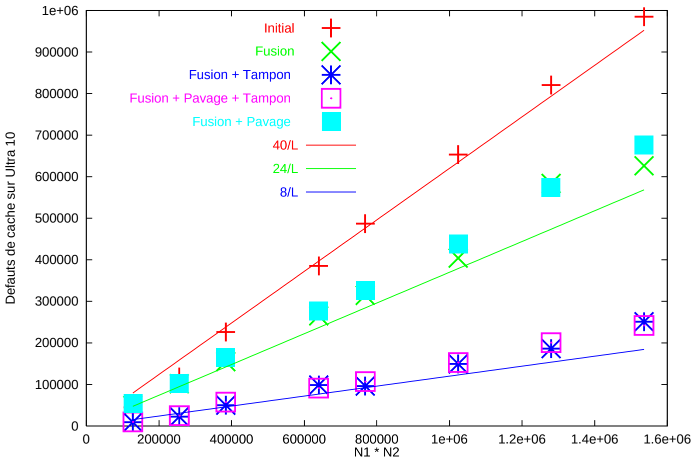


Figure 7.17: Défauts de cache externe (L2) sur *Ultra 10* pour les différentes transformations du code de l'exemple en *figure 7.31*

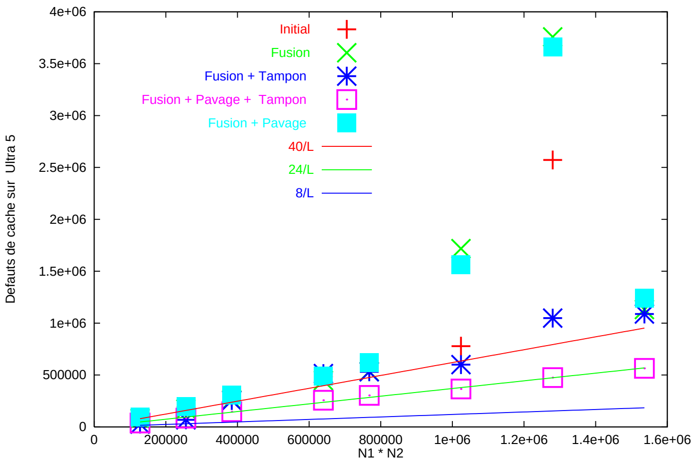


Figure 7.18: Défauts de cache externe (L2) sur *Utra 5* pour les différentes transformations du code de l'exemple en *figure 7.31*

les défauts de caches causés par toutes les transformations suivent asymptotiquement des droites théoriques bien définies qui seront détaillées dans la section suivante. Nous pouvons remarquer aussi que (1) la fusion avec allocation des tampons et (2) la fusion avec pavage et avec allocation des tampons donnent pratiquement les même résultats. Sur *Utra 5*, seul la fusion avec pavage et avec allocation des tampons suit asymptotiquement une droite théorique bien définie. Cette transformation donne le meilleur résultat, c'est-à-dire le nombre minimum de défauts de cache. Ceci est prévisible car la taille du cache externe de cette machine est considérablement inférieure aux caches des autres machines.

Résultats attendus

Le chargement de données dans le cache externe des trois machines précédentes se fait par bloc de taille $L = 64$ octets. Les six matrices utilisées dans le code initial en *figure 7.31* ont une taille $N_1 * N_2$ chacune ; chaque élément de ces matrices est stocké sur 4 octets. Sur *Sun-Blade-1000* nous avons supposé que $1280 \leq N_1 \leq 6400$ et $1200 \leq N_2 \leq 3000$ et sur

Utra 10 et *Ultra 5* nous avons supposé que $640 \leq N_1 \leq 1280$ et $200 \leq N_2 \leq 1200$. Ces six matrices ne peuvent pas tenir en même temps dans le cache, car la taille en octet de ces six matrices est $24 * N_1 * N_2$ octets et nous avons :

1. $640 \leq N_1 \leq 1280$ et $200 \leq N_2 \leq 1200$ dans le cas de *Utra 10* et *Utra 5*. Par conséquent $24 * N_1 * N_2$ octets $> 2\ M$ octets $> 256\ k$ octets.
2. $1280 \leq N_1 \leq 6400$ et $1200 \leq N_2 \leq 3000$ dans le cas de *Sun-Blade-1000*. Par conséquent $24 * N_1 * N_2\ o > 8\ M$ octets.

Par contre, le cache peut contenir plusieurs lignes d'une seule matrice car la taille d'une ligne est $4 * N_2$ octets et nous avons :

1. $200 \leq N_2 \leq 1200$ dans le cas de *Utra 5* et *Ultra 10*. Par conséquent $4 * N_2$ octets $\leq 4.8\ k$ octets $<< 256\ k$ octets.
2. $1200 \leq N_2 \leq 3000$ dans le cas de *Sun-Blade-1000*. Par conséquent $4 * N_2$ octets $\leq 12\ k$ octets $<< 8\ M$ octets.

Le parcours contigu des éléments d'une matrice en écriture ou en lecture engendre donc naturellement $\frac{4*N_1*N_2}{L}$ défauts de cache.

1) Code initial

Le code initial en *figure 7.31* est constitué de cinq nids. Les écritures des cinq nids engendrent $\frac{4*N_1*N_2}{L}$ défauts de cache chacune. Les lectures de ces cinq nids engendrent aussi $\frac{4*N_1*N_2}{L}$ défauts de cache chacune. En effet, en régime permanent, pour exécuter toutes les itérations j d'un i donné :

1. nous avons besoin des lignes $i-1$, i et $i+1$ dans le cas des nids 1, 3 et 5. Les lignes $i-1$ et i sont déjà chargées précédemment dans le cache. Il n'y a que la ligne $i+1$ qui n'est pas disponible ; le chargement de cette ligne engendre $\frac{4*N_2}{L}$ défauts de cache. Par conséquent, l'exécution de toutes les itérations i engendre $\frac{4*N_1*N_2}{L}$ défauts de cache.
2. nous avons besoin de la ligne i dans le cas des nids 2 et 4, le chargement de cette ligne engendre $\frac{4*N_2}{L}$ défauts de cache. Par conséquent, l'exécution de toutes les itérations i engendre $\frac{4*N_1*N_2}{L}$ défauts de cache.

Le nombre total de défauts de cache engendré par le code initial est alors :

$$\frac{40 * N_1 * N_2}{L}$$

Ce nombre est représenté par la droite de pente $\frac{40}{L}$ sur les *figures* 7.16, 7.17 et 7.18.

2) Code fusionné

Dans le code fusionné, les écritures des cinq instructions engendrent $\frac{4*N_1*N_2}{L}$ défauts de cache chacune. Par contre dans le cas de la lecture, seule la première instruction engendre $\frac{4*N_1*N_2}{L}$ défauts de cache. Par conséquent, le nombre total de défauts de cache engendré par le nid fusionné est :

$$\frac{24 * N_1 * N_2}{L}$$

Ce nombre est représenté par la droite de pente $\frac{24}{L}$ sur les *figures* 7.16, 7.17 et 7.18.

3) Code fusionné avec allocation de tampon

Dans le code fusionné avec allocation des tampons, nous avons deux tableaux IN et OUT de taille $4 * N_1 * N_2$ chacun et 4 tampons B_1, B_2, B_3 et B_4. Les tailles de ces tampons sont respectivement 12, $4 * (2N_2 - 1)$, 4 et $4 * (2N_2 - 5)$. Le nombre de défauts de cache engendré par B_1, B_2, B_3 et B_4 sont respectivement $\frac{12}{L}$, $\frac{4*(2N_2-1)}{L}$, $\frac{4}{L}$ et $\frac{4*(2N_2-5)}{L}$. L'écriture dans la matrice OUT et la lecture dans la matrice IN engendre $\frac{4*N_1*N_2}{L}$ défauts de cache chacune. Par conséquent, le nombre total de défauts de cache engendré par le nid fusionné avec allocation des tampons est :

$$\frac{8 * N_1 * N_2}{L} + \frac{(16 * N_2 - 8)}{L} \simeq \frac{8 * N_1 * N_2}{L}$$

Ce nombre est représenté par la droite de pente $\frac{8}{L}$ sur les *figures* 7.16, 7.17 et 7.18.

4) Fusion avec pavage

Comme nous avons supposé dans le cas de la fusion que les lignes $i - 1$, i et $i + 1$ de n'importe quel tableau se trouvent toujours dans le cache externe à toute itération $(i,\ j)$, alors la fusion avec pavage a le même résultat que la fusion. C'est-à-dire que le nombre total de défauts de cache engendré par le nid fusionné avec pavage est :

$$\frac{24 * N_1 * N_2}{L}$$

Ce nombre est représenté par la droite de pente $\frac{24}{L}$ sur les *figures* 7.16, 7.17 et 7.18.

5) Fusion avec pavage et avec allocation des tampons

Dans le code fusionné avec pavage et avec allocation des tampons, nous avons deux tableaux IN et OUT de taille $4 * N_1 * N_2$ chacun et 12 tampons $B_{1,1}$, $B_{1,2}$, $B_{1,3}$, $B_{2,1}$, $B_{2,2}$, $B_{2,3}$, $B_{3,1}$, $B_{3,2}$, $B_{3,3}$, $B_{4,1}$, $B_{4,2}$ et $B_{4,3}$. Les tailles de ces tampons sont respectivement 0, $16 * T_1$, $4 * T_2$, $16 * (N_2 - 2)$, $16 * T_1$, $12 * T_2$, 0, 0, $4 * T_2$, $16 * (N_2 - 4)$, $16 * T_1$ et $12 * T_2$. Les coefficients T_1 et T_2 sont les tailles des tuiles respectivement sur les axes i et j. Le nombre total de défauts de cache est la somme des défauts de cache de tous ces tampons et des tableaux IN et OUT, qui est donné par :

$$\frac{8 * N_1 * N_2}{L} + \frac{32 * N_2 + 48 * T_1 + 32 * T_2 - 96}{L} \simeq \frac{8 * N_1 * N_2}{L}$$

Ce nombre est représenté par la droite de pente $\frac{8}{L}$ sur les *figures* 7.16, 7.17 et 7.18.

7.4.2 Discutions des résultats théoriques et expérimentaux

En analysant les résultats théoriques précédents nous pouvons remarquer que :

1. La fusion et la fusion avec pavage ont pratiquement le même résultat théorique. Cette égalité est confirmée par les expériences qui sont faites sur *Sun-Blade-1000* et *Utra 10* (*figures* 7.16 et 7.17).
2. La fusion avec allocation des tampons et la fusion avec pavage et avec allocation des tampons ont pratiquement le même résultat théorique. Cette égalité est confirmée aussi par les expériences qui sont faites sur *Sun-Blade-1000* et *Utra 10* (*figures* 7.16 et 7.17).

Sur *Utra 5*, notre modèle théorique ne correspond pas aux résultats expérimentaux, car la taille du cache externe de cette machine est inférieure aux tailles des caches des deux autres machines.

7.4.3 Temps d'exécution

Les *figures* 7.19, 7.20 et 7.21 donnent les temps d'exécutions en fonction de la taille des tableaux respectivement sur *Sun-Blade-1000*, *Utra 10* et *Ultra 5*.

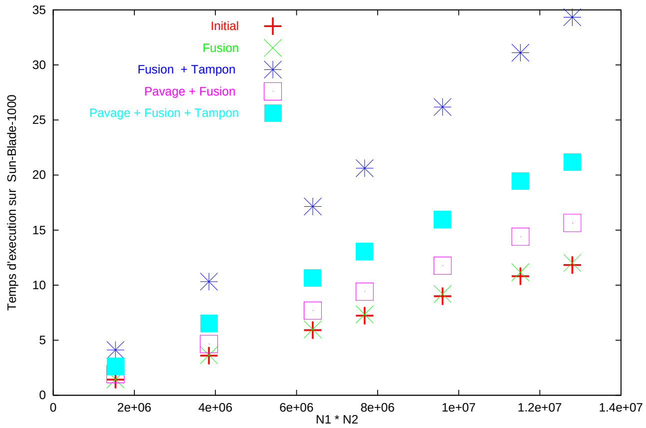


Figure 7.19: Temps d'exécution des différentes transformations du code en *figure 7.31* sur *Sun-Blade-1000*

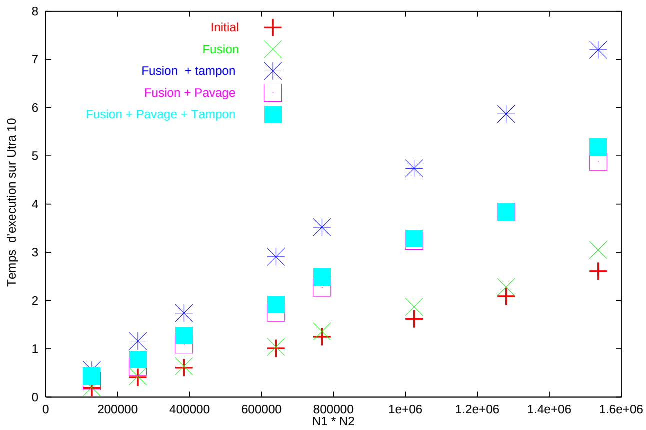


Figure 7.20: Temps d'exécution des différentes transformations du code en *figure 7.31* sur *Utra 10*

Comme pour le cas d'école, la fusion avec allocation des tampons augmente considérablement le temps d'exécution. Nous avons signalé précédemment que, si les tailles des tampons sont des puissances de 2, le temps d'exécution du code fusionné avec allocation des tampons est égal à celui du code initial tandis que le nombre de défauts de cache reste inchangé. La *figure 7.22* montre la réduction du temps d'exécution du code fusionné avec allocation de tampons de tailles égales à la première puissance de 2 supérieure au volume réellement nécessaire. La *figure 7.23* donne les temps d'exécutions des différentes transformations du code en *figure 7.31* avec les tailles des tampons dans le cas de la fusion avec allocation des

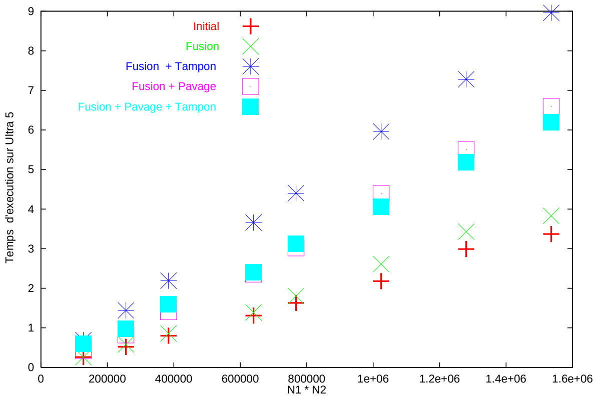


Figure 7.21: Temps d'exécution des différentes transformations du code en *figure 7.31* sur *Utra 5*

tampons sont égales à une puissance de 2. Sur les deux autres machines nous avons la même réduction du temps d'exécution.

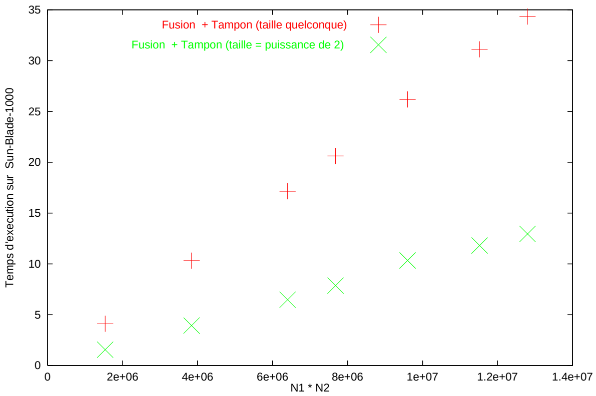


Figure 7.22: Réduction du temps d'exécution par des tailles des tampons égales à des puissances de 2 sur *Sun-Blade-1000*

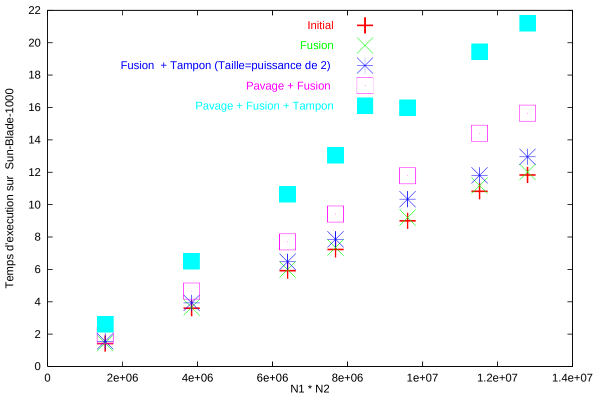


Figure 7.23: Temps d'exécution des différentes transformations du code en *figure 7.31* sur *Sun-Blade-1000* (Tailles des tampons sont égales à des puissances de 2)

7.4.4 Taille mémoire

Comme le cas d'école, nous donnons les tailles mémoires en octet des différents tableaux et tampons utilisés dans les différentes transformations.

1) Code initial :

Dans le code initial nous avons 6 tableaux ; chacun de ces tableaux a une taille de $4 * N_1 * N_2$. Par conséquent, le volume mémoire de tous les tableaux manipulés dans ce code est :

$$24 * N_1 * N_2$$

2) Code fusionné :

Le nombre de tableaux utilisés dans ce code est le même que celui utilisé dans le code initial. Le volume mémoire de tous les tableaux manipulés dans ce code est donc :

$$24 * N_1 * N_2$$

3) Code fusionné avec allocation des tampons :

Dans ce code nous avons deux tableaux de taille $4 * N_1 * N_2$ chacun et 4 tampons B_1, B_2, B_3 et B_4. Les tailles de ces tampons sont respectivement 12, $4 * (2N_2 - 1)$, 4 et $4 * (2N_2 - 5)$. le volume mémoire de tous les tableaux et les tampons manipulés dans ce code est donc :

$$8 * N_1 * N_2 + 16 * N_2 - 8$$

4) Code fusionné avec pavage :

Le nombre de tableaux utilisés dans ce code est le même que celui utilisé dans le code initial. Le volume mémoire de tous les tableaux manipulés dans ce code est donc :

$$24 * N_1 * N_2$$

5) Code fusionné avec pavage et avec allocation des tampons :

Dans ce code nous avons deux tableaux de taille $4 * N_1 * N_2$ chacun et 12 tampons $B_{1,1}$, $B_{1,2}$, $B_{1,3}$, $B_{2,1}$, $B_{2,2}$, $B_{2,3}$, $B_{3,1}$, $B_{3,2}$, $B_{3,3}$, $B_{4,1}$, $B_{4,2}$ et $B_{4,3}$. Les tailles de ces tampons sont respectivement 0, $16 * T_1$, $4 * T_2$, $16 * (N_2 - 2)$, $16 * T_1$, $12 * T_2$, 0, 0, $4 * T_2$, $16 * (N_2 - 4)$, $16 * T_1$ et $12 * T_2$. Les coefficients T_1 et T_2 sont les tailles des tuiles respectivement sur les axes i et j. Le volume mémoire de tous les tableaux et les tampons manipulés dans ce code est donc :

$$8 * N_1 * N_2 + 32 * N_2 + 48 * T_1 + 32 * T_2 - 96$$

7.4.5 Taille des codes

La partie du code qui est différente entre le code initial et ses différentes transformations est la fonction *TRAITE* qui contient les calculs des cinq nids. Le tableau suivant donne les tailles du code initial et ses différentes transformations.

Les codes	Taille du code *Fortran*	Taille du code *Assembleur*
Initial	29	317
Fusion	16	285
Fusion + Tampon	21	517
Fusion + Pavage	53	569
Fusion + Pavage + Tampon	556	5816

Comme l'exemple d'école, c'est le code fusionné avec pavage et avec allocation des tampons qui possède la plus grande taille. Cela est toujours dû à la même raison : les tests ajoutés pour savoir dans quel tampon nous sauvegardons et nous récupérons la donnée du tableau correspondant.

7.4.6 Défauts de TLB

Dans les systèmes embarqués nous n'avons pas de mémoire virtuelle, par conséquent nous n'avons pas de défauts de TLB. Nous avons néanmoins mesuré les défauts de TLB de données pour les différentes transformations du code en *figure 7.31* qui sont données en *figure 7.24*. L'allocation de tampons dans le code fusionné et dans le code fusionné avec pavage permet de réduire d'une manière considérable le nombre de défauts de *DATA TLB* et le meilleur résultat est obtenu par le code fusionné avec allocation des tampons.

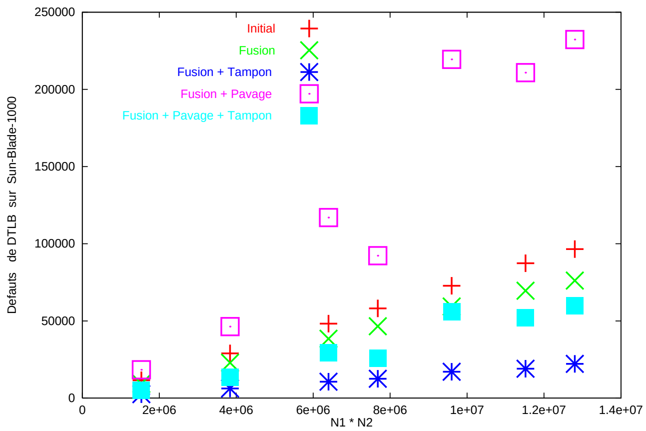


Figure 7.24: Défauts de DATA TLB des différentes transformations du code en *figure 7.31* sur *Sun-Blade-1000*

7.5 Conclusion

À l'exception de la fusion de boucles avec pavage à deux niveaux et avec le remplacement de chaque tableau par plusieurs tampons et registres, toutes les transformations détaillées dans les chapitres précédents sont implantées dans *Pips* [64]. Nos expériences montrent que nos techniques d'allocation des tampons permettent de réduire d'un facteur 5 le nombre de défauts de cache externe ce qui était notre objectif. Pour réduire les temps d'exécution, nous proposons (1) d'éliminer les gardes sur les différentes instructions par le découpage du nid fusionné et tuilé en plusieurs sous-nids et (2) de supprimer entièrement les *modulos* par le déroulage de boucles. Ceci fera l'objet d'autres travaux.

```
INTEGER function ma(x,y,z,l,m)
INTEGER x, y,z,l,m
ma = x+y+z+l+m
END
subroutine traite(IN, OUT)
INTEGER IN(1000,1000), OUT(1000,1000)
INTEGER B(1000,1000)
EXTERNAL MA
INTEGER MA
do i=5, 996
   do j=5,996
     B(j,i)=ma(IN(j,i-4),IN(j-4,i),IN(j,i),IN(j+4,i),IN(j,i+4))
   enddo
enddo
do i=9, 992
   do j=9,992
     OUT(j,i)=ma(B(j,i-4),B(j-4,i),B(j,i),B  (j+4,i),B(j,i+4))
   enddo
enddo
end
program ECOLE
INTEGER A(1000,1000),B(1000,1000)
do i=1, 1000
   do j=1,1000
     A(j,i)=2*i+j
   enddo
enddo
call traite(A, B)
end
```

Figure 7.25: Code en entrée (Cas d'Ecole)

```
INTEGER*4 FUNCTION MA(X,Y,Z,L,M)
INTEGER X,Y,Z,L,M
MA = X+Y+Z+L+M
END
SUBROUTINE TRAITE(IN,OUT)
INTEGER*4 MA
EXTERNAL MA
INTEGER I,J
INTEGER IN(1 :1000,1 :1000),OUT(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I = 1, 992
   DO J = 5, 996
     B(J,4+I) = MA(IN(J,4+I-4), IN(J-4,4+I), IN(J,4+I), IN(J+4,4+I), IN(J,4+I+4))
     IF (I.GE.9.AND.J.GE.9.AND.J.LE.992) OUT(J,I) = MA(B(J,I-4), B(J-4,I), B(J,I), B(J+4,I), B(J,I+4))
   ENDDO
ENDDO
END
PROGRAM ECOLE
INTEGER I,J
INTEGER A(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I = 1, 1000
   DO J = 1, 1000
     A(J,I) = 2*I+J
   ENDDO
ENDDO
call traite(A, B)
END
```

Figure 7.26: Fusion du code en *figure 7.25*

```
INTEGER*4 FUNCTION MA(X,Y,Z,L,M)
INTEGER X,Y,Z,L,M
MA = X+Y+Z+L+M
END
SUBROUTINE TRAITE(IN,OUT)
INTEGER*4 MA
EXTERNAL MA
INTEGER I,J
INTEGER B1(0 :7936),IN(1 :1000,1 :1000),OUT(1 :1000,1 :1000)
DO I = 1, 992
   DO J = 5, 996
     B1(MOD(J-997+992*I, 7937)) = MA(IN(J,4+I-4), IN(J-4,4+I), IN(J,4+I), IN(J+4,4+I), IN(J,4+I+4))
     IF (I.GE.9.AND.J.GE.9.AND.J.LE.992) OUT(J,I) = MA(B1(MOD(992*I-8933+J, 7937)), B1(MOD(992*I-4969+J, 7937)),
    B1(MOD (992*I-4965+J, 7937)), B1(MOD(992*I-4961+J, 7937)), B1( MOD(992*I-997+J, 7937)))
   ENDDO
ENDDO
END
PROGRAM ECOLE INTEGER I,J
INTEGER A(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I = 1, 1000
   DO J = 1, 1000
     A(J,I) = 2*I+J
   ENDDO
ENDDO
call traite(A, B)
END
```

Figure 7.27: Fusion avec allocation d'un tampon du code en *figure 7.25*

```
INTEGER*4 FUNCTION MA(X,Y,Z,L,M)
INTEGER X,Y,Z,L,M
MA = X+Y+Z+L+M
END
SUBROUTINE TRAITE(IN,OUT)
INTEGER*4 MA
EXTERNAL MA
INTEGER I1,I2,J1,J2,R2.0,R2.1,R2.2,R2.3,R2.4
INTEGER B1.3(0 :399,0 :8),B1.2(0 :7,0 :399,0 :1),B1.1(0 :991,0 :7,0 :1),IN(1 :1000,1 :1000),OUT(1 :1000,1 :1000)
DO I1 = 0, 2
   DO J1 = 0, 2
     DO I2 = 0, 399
       DO J2 = 0, 399
         IF (400*I1+I2.LE.991.AND.400*J1+J2.LE.991) THEN
           B1.3(J2,MOD(I2, 9)) = MA(IN(5+400*J1+J2,5+400*I1 +I2-4), IN(5+400*J1+J2-4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2),
           IN(5+400*J1+J2+4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2+4))
           IF (-392+J2.GE.0) B1.2(-392+J2,I2,MOD(J1, 2)) = MA(IN(5+400*J1+J2,5+400*I1+I2-4), IN(5+400*J1+J2-4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2),
           IN(5+400*J1+J2+4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2+4))
           IF (-392+I2.GE.0) B1.1(400*J1+J2,-392+I2,MOD(I1, 2)) = MA(IN(5+400*J1+J2,5+400*I1+I2-4), IN(5+400*J1+J2-4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2),
           IN(5+400*J1+J2+4,5+400*I1+I2), IN(5+400*J1+J2,5+400*I1+I2+4))
         ENDIF
         IF (400*I1+I2.GE.8.AND.400*I1+I2.LE.991.AND.400*J1+J2.GE.8.AND.400*J1+J2.LE.991) THEN
           IF (-8+I2.GE.0.AND.-8+J2.GE.0) THEN
             OUT(1+400*J1+J2,1+400*I1+I2) = MA(B1.3(-4+J2, MOD(-8+I2, 9)), B1.3(-8+J2,MOD(-4+I2, 9)),B1.3(-4+J2,MOD(-4+I2, 9)),
             B1.3(J2,MOD(-4+I2 , 9)), B1.3(-4+J2,MOD(I2, 9)))
           ELSE
             IF (-8+I2.LT.0) THEN
               R2.0 = B1.1(400*J1+-4+J2,-392+392+I2,MOD( -1+I1, 2))
             ELSEIF (-4+J2.LT.0) THEN
               R2.0 = B1.2(-392+396+J2,-8+I2,MOD(-1+J1, 2 ))
             ELSE
               R2.0 = B1.3(-4+J2,MOD(-8+I2, 9))
             ENDIF
             IF (-4+I2.LT.0) THEN
               R2.1 = B1.1(400*J1+-8+J2,-392+396+I2,MOD(-1+I1, 2))
             ELSEIF (-8+J2.LT.0) THEN
               R2.1 = B1.2(-392+392+J2,-4+I2,MOD(-1+J1, 2))
             ELSE
               R2.1 = B1.3(-8+J2,MOD(-4+I2, 9))
             ENDIF
             IF (-4+I2.LT.0) THEN
               R2.2 = B1.1(400*J1+-4+J2,-392+396+I2,MOD(-1+I1, 2))
             ELSEIF (-4+J2.LT.0) THEN
               R2.2 = B1.2(-392+396+J2,-4+I2,MOD(-1+J1, 2))
             ELSE
               R2.2 = B1.3(-4+J2,MOD(-4+I2, 9))
             ENDIF
             IF (-4+I2.LT.0) THEN
               R2.3 = B1.1(400*J1+J2,-392+396+I2,MOD(-1+I1, 2))
             ELSEIF (J2.LT.0) THEN
               R2.3 = B1.2(-392+400+J2,-4+I2,MOD(-1+J1, 2))
             ELSE
               R2.3 = B1.3(J2,MOD(-4+I2, 9))
             ENDIF
             IF (I2.LT.0) THEN
               R2.4 = B1.1(400*J1+-4+J2,-392+400+I2,MOD( -1+I1, 2))
             ELSEIF (-4+J2.LT.0) THEN
               R2.4 = B1.2(-392+396+J2,I2,MOD(-1+J1, 2))
             ELSE
               R2.4 = B1.3(-4+J2,MOD(I2, 9))
             ENDIF
             OUT(1+400*J1+J2,1+400*I1+I2) = MA(R2.0, R2.1 , R2.2, R2.3, R2.4)
           ENDIF
         ENDIF
       ENDDO
     ENDDO
   ENDDO
ENDDO
END
PROGRAM ECOLE
INTEGER I,J
INTEGER A(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I = 1, 1000
   DO J = 1, 1000
     A(J,I) = 2*I+J
   ENDDO
ENDDO
call traite(A, B)
END
```

Figure 7.28: Fusion avec pavage à un seul niveau et avec allocation des tampons du code en *figure 7.25*

```
INTEGER*4 FUNCTION MA(X,Y,Z,L,M)
INTEGER X,Y,Z,L,M
MA = X+Y+Z+L+M
END
SUBROUTINE TRAITE(IN,OUT)
INTEGER*4 MA
EXTERNAL MA
INTEGER I1,I2,I3,J1,J2,J3
INTEGER IN(1 :1000,1 :1000),OUT(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I1 = 0, 2
   DO J1 = 0, 2
     DO I2 = 0, 199
       DO J2 = 0, 199
         DO I3 = 0, 1
           DO J3 = 0, 1
             IF (400*I1+2*I2+I3.LE.991.AND.400*J1+2*J2+J3 .LE.991) B(5+400*J1+2*J2+J3,5+400*I1+2*I2+I3) = MA(IN(5+400*J1+2*J2+J3,5+400*I1+2*I2+I3-4) ,
             IN(5+400*J1+2*J2+J3-4,5+400*I1+2*I2+I3), IN(5+400*J1+2*J2+J3,5+400*I1+2*I2+I3), IN(5+400 *J1+2*J2+J3+4,5+400*I1+2*I2+I3),
             IN(5+400*J1+2*J2+J3,5+400*I1+2*I2+I3+4))
             IF (400*I1+2*I2+I3.GE.8.AND.400*I1+2*I2+I3 .LE.991.AND.400*J1+2*J2+J3.GE.8.AND.400*J1+2* J2+J3.LE.991)
             OUT(1+400*J1+2*J2+J3,1+400*I1+2 *I2+I3) = MA(B(1+400*J1+2*J2+J3,1+400*I1+2*I2 +I3-4), B(1+400*J1+2*J2+J3-4,1+400*I1+2*I2+I3 ),
             B(1+400*J1+2*J2+J3,1+400*I1+2*I2+I3), B(1+ 400*J1+2*J2+J3+4,1+400*I1+2*I2+I3), B(1+400* J1+2*J2+J3,1+400*I1+2*I2+I3+4))
           ENDDO
         ENDDO
       ENDDO
     ENDDO
   ENDDO
ENDDO
END
PROGRAM TILING5
INTEGER I,J
INTEGER A(1 :1000,1 :1000),B(1 :1000,1 :1000)
DO I = 1, 1000
   DO J = 1, 1000
     A(J,I) = 2*I+J
   ENDDO
ENDDO
call traite(A, B)
END
```

Figure 7.29: Fusion avec pavage à deux niveaux du code en *figure 7.25*

do (i' = 0, i' = 5)
  do ($j' = 0$, $j' = \frac{i'}{6} - \frac{1}{6}$)
    $S = S + A(j', i' - 6 * j') * X(i' - 6 * j')$
  enddo
  if (**entier** $\frac{i'}{6}$) **then** { S=0 ; $S = S + A(\frac{i'}{6}, 0) * X(0)$ }
 enddo
do (i' = 6, i' = 24)
  if (**entier** $\frac{i'-6}{6}$) **then** { $S = S + A(\frac{i'-6}{6}, 6) * X(6)$; $Y((\frac{i'-6}{6}) = S$}
  do ($j' = \frac{i}{6}$, $j' = \frac{i'}{6} - \frac{1}{6}$)
    $S = S + A(j', i' - 6 * j') * X(i' - 6 * j')$
  enddo
  if (**entier** $\frac{i'}{6}$) **then** { S=0 ; $S = S + A(\frac{i'}{6}, 0) * X(0)$ }
  enddo
do (i' = 25, i' = 30)
  if (**entier** $\frac{i'-6}{6}$) **then** { $S = S + A(\frac{i'-6}{6}, 6) * X(6)$; $Y(\frac{i'-6}{6}) = S$ }
  do ($j' = \frac{i}{6}$, $j' = 4$)
    $S = S + A(j', i' - 6 * j') * X(i' - 6 * j')$
  enddo
enddo

Figure 7.30: Produit matrice vecteur : Code généré dans Pips

```
INTEGER function ma1(x,y,z)
INTEGER x, y,z
ma1= ( 68*x+99*y+68*z)/ (2*68+99)
END
INTEGER function ma2(x,y,z,l,m,o,p,q,r)
INTEGER x, y,z, l,m,o,p,q,r
ma2= max(abs(x-m),abs(y-m),abs(z-m),abs(l-m),abs(r-m),abs(o-m) ,abs(p-m),abs(q-m))
END
INTEGER function ma3(x)
INTEGER x
ma3= 255-x
END
INTEGER function ma4(x,y,z,l,m,o,p,q,r)
INTEGER x, y,z,l, m,o,p,q,r
ma4= 255
if ((x.gt.m).or.(y.gt.m).or.(z.gt.m).or.(l.gt.m).or.(r.gt.m) .or.(o.gt.m).or.(p.gt.m).or.(q.gt.m)) then
ma4=0
endif
END
subroutine traite(IN, OUT)
INTEGER IN(200,640), Tem(200,640), Gim(200,640), Cim(200,640), OUT(200,640),Tem2(200,640)
do i=2, 639
   do j=1, 200
     Tem(j,i)=ma1(IN(j,i-1),IN(j,i),IN(j,i+1))
   enddo
enddo
do i=2, 639
   do j=2, 199
     Gim(j,i)=ma1(Tem(j-1,i),Tem(j,i),Tem(j+1,i))
   enddo
enddo
do i=3,638
   do j=3,198
     Cim(j,i)=ma2(Gim(j-1,i-1),Gim(j,i-1),Gim(j+1,i-1), Gim(j-1,i), Gim(j,i),Gim(j+1,i), Gim(j-1,i+1),Gim(j,i+1), Gim(j+1,i+1))
   enddo
enddo
do i= 3,638
   do j= 3,198
     Tem2(j,i)=ma3( Cim(j,i))
   enddo
enddo
do i=4, 637
   do j=4,197
     OUT(j,i)= ma4(Tem2(j-1,i-1),Tem2(j,i-1),Tem2(j+1,i-1), Tem2(j-1,i), Tem2(j,i),Tem2(j+1,i), Tem2(j-1,i+1), Tem2(j,i+1), Tem2(j+1,i+1))
   enddo
enddo
end
program tiling5
INTEGER INim(200,640), Outim(200,640)
call readimage(Inim, "Imagein/200-640.pgm")
call traite(Inim, Outim)
call writeimage(Outim,"Imageout/200-640.pgm")
end
```

Figure 7.31: Applicaton réelle : Détection de cavités

Chapitre 8

Conclusions et perspectives

Les travaux présentés dans cette thèse ont pour objectif final la réduction de l'énergie nécessaire à des applications de traitement du signal exécutées sur des systèmes embarqués tels que les téléphones cellulaires ou les caméras vidéo numériques. La réduction de la consommation électrique nécessite principalement une diminution du nombre d'accès à la mémoire *off-chip* du processeur comme l'ont montré les travaux de l'IMEC. En terme de compilation, cette diminution passe par une augmentation de la localité des données du code.

Nous avons étudié plusieurs transformations de programmes permettant d'augmenter la localité. Nous les avons étendues et améliorées. Puis nous les avons implantées dans le prototype de compilateur source-à-source *Pips* et nous avons effectué des mesures sur diverses machines cibles. Ces mesures montrent que le nombre d'accès mémoire peut être réduit par un facteur allant jusqu'à 3. La plupart des travaux dans le domaine des transformations de programmes qui sont consacrés aux pavages et aux transformations unimodulaires sont destinés à des codes particuliers qui sont constitués d'un seul nid de boucles parfaitement imbriquées. Les codes scientifiques sont en général plus compliqués. Par exemple, les codes des applications de traitement du signal sont des séquences de nids de boucles. Dans cette thèse, nous avons présenté de nouvelles techniques de transformations de programmes qui prennent en entrée des codes plus généraux qui sont des séquences de nids de boucles et des nids de boucles mal imbriquées.

8.1 Détail des Contributions

Nous avons étudié trois techniques d'augmentation de la localité des données utilisant simultanément un réordonnancement temporel des calculs et une réallocation spatio-temporelle des données. Ces techniques sont : la fusion avec allocation des tampons, le pavage avec allocation des tampons et l'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées.

8.1.1 Fusion avec allocation des tampons

Dans le cas général, avant d'appliquer la fusion sur deux nids de boucles, nous devons nous assurer qu'ils ont le même domaine d'itération. Dans le *chapitre 3*, nous avons appliqué la fusion sur une séquence de nids de boucles qui n'ont pas nécessairement le même domaine d'itération. La généralisation de cette méthode à d'autres classes de codes nécessite une étude plus détaillée, car le problème de la fusion de boucles dans le cas général est

NP_ complet. Ensuite, nous avons présenté une nouvelle technique d'allocation des tampons qui consiste à remplacer chaque tableau intermédiaire par un tampon circulaire et séquentiel qui ne contiendra que les données vivantes, et à chacun de ces tampons nous avons associé une fonction d'accès qui permet de sauvegarder et de récupérer les données du tableau correspondant. Cette méthode d'allocation des tampons permet de réduire d'une manière considérable les défauts de cache.

8.1.2 Pavage hiérarchique avec allocation des tampons

Dans la littérature, le pavage est à un seul niveau. Il est appliqué généralement sur un seul nid de boucles et un seul ordonnancement des tuiles et des itérations à l'intérieur de celles-ci est pris en considération.

Dans le *chapitre 4*, nous avons présenté une méthode qui consiste à appliquer la fusion et le pavage à un seul niveau simultanément sur une séquence de nids de boucles. Notre pavage est représenté par deux matrices : (1) une matrice de pavage $\mathcal{A}$ qui permet de spécifier la taille des tuiles et (2) une matrice de permutation $\mathcal{P}$ qui permet de spécifier l'ordre d'exécution des itérations. Grâce à la matrice de permutation $\mathcal{P}$, nous avons pris en considération tous les ordonnancements possibles (ordonnancement des tuiles et des itérations à l'intérieur de ces tuiles) et nous les avons modélisés par une seule transformation.

Pour maintenir en mémoire les données vivantes, nous avons présenté deux techniques d'allocation de tampons. La première méthode est similaire à celle présentée dans la fusion, c'est-à-dire que nous avons remplacé chaque tableau intermédiaire par un tampon circulaire et séquentiel qui ne contient (presque) que les données vivantes. Ceci présente les inconvénients suivants : (1) à certaines itérations le nombre de données mortes stockées dans ce tampon est élevé et (2) ses fonctions d'adressages sont compliquées. Pour pallier les inconvénients de la première méthode, nous avons divisé dans la deuxième méthode l'ensemble des données vivantes de chaque tableau en plusieurs sous-ensembles et attribué un tampon à chacun de ces derniers. Dans le *chapitre 5*, nous avons appliqué la fusion et le pavage à deux niveaux sur une séquence de nids de boucles et nous avons présenté une nouvelle méthode d'allocation des tampons et des registres. Notre pavage est toujours représenté par deux matrices $\mathcal{A}$ et $\mathcal{P}$ et il est modélisé comme une transformation de boucles. Dans les deux cas (*chapitre 4* et *chapitre 5*) nous avons mis en relation les différents paramètres et nous avons donné deux solutions pour les vecteurs de décalage pour que l'application du pavage avec fusion soit valide.

8.1.3 Élimination des gardes

L'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées a été traitée par peu d'auteurs. Nous avons traité les deux problèmes en détail : (1) la mise sous forme parfaitement imbriquée d'un nid de boucles et (2) l'élimination de gardes dans le nid transformé. Pour la première partie, nous avons donné la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid parfaitement imbriqué. Si cette condition n'est pas satisfaite, ou bien (1) la boucle interne est vide et nous éliminons entièrement le corps de la boucle interne, ou (2) nous décomposons le nid de boucles en une séquence de deux nids de boucles. Dans le premier nid, nous faisons rentrer toutes les instructions dans la boucle la plus interne et dans le second nous éliminons le corps de la boucle interne pour ne garder que les instructions mal imbriquées.

Pour la seconde partie du problème, nous avons proposé un algorithme qui permet d'éliminer les gardes, en décomposant le nid transformé en une séquence de nids de boucles. Notre algorithme minimise le nombre de nids de boucles nécessaires à l'élimination de toutes les gardes internes. Il se peut que des gardes subsistent dans le corps de la boucle externe. Elles pourraient aussi être éliminées par un déroulage de cette boucle, qui aurait l'inconvénient d'augmenter la taille du code. Ces contributions sont comparées à l'état de l'art dans la *Section 5.6*. Le code que nous générons est le plus compact et le plus lisible.

Vu la difficulté de la deuxième partie de ce problème, nous avons limité notre étude aux nids de boucles à deux dimensions, non paramétriques, parce que le nombre de formes que peut prendre l'algorithme décomposé est déjà très élevé et l'expansion de code non négligeable dans les cas les plus défavorables. Dans de prochains travaux, nous espérons pouvoir généraliser cette étude au cas général, paramétrique et en dimension quelconque, mais la taille du code produit risque d'exploser. Il sera sans doute nécessaire de se restreindre aux boucles les plus internes pour lesquelles l'élimination de garde permet d'appliquer un algorithme de pipeline logiciel et produit de toute façon la partie essentielle de l'optimisation puisque ce sont les tests les plus fréquemment exécutés.

8.1.4 Implantations et mesures expérimentales

Toutes nos techniques ont été implantées dans $PIPS$. Les mesures expérimentales montrent que les différentes techniques permettent de réduire de manière considérable les défauts de cache et donc de réduire la consommation de l'énergie électrique.

8.2 Perspectives et travaux futurs

Les techniques présentées dans cette thèse sont appliquées à des codes particuliers qui sont, soit des séquences de nids de boucles de profondeur quelconque, soit un nid de boucles mal imbriquées de profondeur 2. Dans les travaux futurs, il serait intéressant de généraliser toutes ces techniques à des codes quelconques.

8.2.1 Fusion

La généralisation de l'application de la fusion avec allocation des tampons sur des codes quelconques nécessite un effort supplémentaire car le problème de la fusion dans le cas général est *NP_ complet* et il est difficile de savoir quels sont les tableaux que nous pouvons remplacer par des tampons. Il serait intéressant d'étudier d'autres méthodes d'allocations de tampons et de les combiner avec l'allocation des registres.

8.2.2 Pavage

De la même manière que pour la fusion, il serait intéressant de généraliser l'application du pavage avec fusion et allocation des tampons à des codes quelconques. Dans cette thèse, nous n'avons considéré que les pavages parallèles aux axes et au maximum deux niveaux de pavage. L'un de nos travaux envisageables dans le futur est de considérer tous les pavages possibles (parallèles et non parallèle aux axes) et plusieurs niveaux de pavage.

8.2.3 Élimination des gardes

Les codes en entrée de cette technique sont des séquences de nids de boucles de profondeur 2 et les bornes des boucles sont des constantes numériques. Nous envisageons de généraliser l'application de cette technique à des nids de boucles mal imbriquées de profondeur quelconque et dont les bornes des boucles sont des paramètres alphanumériques.

Publications personnelles

1. Yu LI and Youcef BOUCHEBABA. A new genetic algorithm for the optimal communication spanning tree problem. Artificial Evolution, 4th European Conference, AE'99, Selected Papers. Lecture Notes in Computer Science 1829 Springer 2000.
2. Yu LI and Youcef BOUCHEBABA. A new genetic algorithm for the optimal communication spanning tree problem. Algotel 2000.
3. Youcef BOUCHEBABA. Application d'un tiling d'E/S sur le produit de deux matrices. Workshop "Compilation et Parallélisation Automatique".
4. Youcef BOUCHEBABA. Elimination des gardes dans les transformées unimodulaires de boucles mal imbriquées. Rapport de Recherche, CRI -2000, A-312.
5. Youcef BOUCHEBABA et Fabien COELHO. Tiling pour une séquence de nids de boucles. Renpar2001.
6. Youcef BOUCHEBABA and Fabien COELHO. Buffered tiling for sequences of loop nests. International workshop on Compilers and Operating Systems for Low Power 2001.
7. Youcef BOUCHEBABA et Fabien COELHO. Pavage pour une séquence de nids de boucles. Revue RSTI-TSI–21/2002. Parallélisme et systèmes distribués, pages 579 à 603.
8. Youcef BOUCHEBABA. Réutilisation de la mémoire pour le pavage. Renpar2002.
9. Tiling and memory reuse for sequences of nested loops. Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings. Lecture Notes in Computer Science 2400 Springer 2002, ISBN 3-540-44049-6.

Bibliographie

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. Phd thesis, University of Illinois at Urbana-Champaign, Novembre 1978.

[2] N. Ahmed, N. Mateev et K. Pingali. « Synthesizing transformations for locality enhancement of imperfectly-nested loop nests ». Dans *Proceedings of the 2000 ACM International Conference on Supercomputing* (Santa Fe, New Mexico, United States, 2000), pp. 141–152.

[3] N. Ahmed, N. Mateev et K. Pingali. « Tiling imperfectly-nested loop nests ». Dans *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (Dallas, Texas, United States, Novembre 2000).

[4] U. Banerjee. *Unimodular transformations of double loops*. Advances in Languages and Compiler for Parallel Processing, chapter 10. The MIT Press, 1991.

[5] U. Banerjee. *Loop transformations for restructuring compilers*. Kluwer Academic Publishers, 1993.

[6] Y. Bouchebaba. « Élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées ». *Rapport interne du CRI : A/312* (2000).

[7] Y. Bouchebaba. « Réutilisation de la mémoire pour le pavage ». Dans *Renpar'14* (Avril 2002).

[8] Y. Bouchebaba et F. Coelho. « Buffered tiling for sequences of loop nests ». Dans *International Workshop on Compilers and Operating Systems for Low Power* (Barcelone, Espagne, 2001).

[9] Y. Bouchebaba et F. Coelho. « Pavage pour une séquence de nids de boucles ». *Journal Technique et science informatiques . Parallélisme et systèmes distribués* **21**, numéro 5 (2002), 579–603.

[10] Y. Bouchebaba et F. Coelho. « Tiling and memory reuse for sequences of nested loops ». Dans *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference. Proceedings. Lecture Notes in Computer Science 2400 Springer 2002* (Paderborn, Germany, Août 2002), pp. 255–264.

[11] Y. Bouchebaba et F. Coelho. « Tiling pour une séquence de nids de boucles ». Dans *Renpar'13* (Paris, Avril 2001).

[12] P. Boulet, A. Darte, G. A. Silber et F. Vivien. « Loop prallelization algorithms : from parallelism extraction to code generation ». *Journal of Parallel Computing* **24**, numéro 3 (1998), 421–444.

[13] P. Boulet et P. Feautrier. « Scanning polyhedra without Do-loops ». Dans *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Paris, Octobre 1998).

[14] D. Callahan. *A Global Approach to Detection of Parallelism*. Phd thesis, Rice University, 1987.

[15] D. Callahan et al. « Estimating interlock and improving balance for pipelined architectures ». *Journal of Parallel and Distributed Computing* **5**, numéro 4 (Août 1988), 334–358.

[16] S. Carr, C. Ding et P. Sweany. « Improving software pipelining with unroll-and-jam ». Dans *Proceedings of the Twenty-Ninth Annual Hawaii International Conference on System Sciences* (Maui HI, 1996), pp. 183–192.

[17] S. Carr et K. Kennedy. « Scalar Replacement in the Presence of Conditional Control Flow ». *Software-Practice and Experience* **1**, numéro 24 (1994), 51–77.

[18] S. Carr, K. S. McKinley et C. W. Tseng. « Compiler optimizations for improving data locality ». Dans *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), pp. 252–262.

[19] F. Cathoor et al. *Custom memory management methodology-Exploration of memory organization for embedded multimedia system design*. Kluwer Academic Publishers, 1988.

[20] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele et H. DeMan. « Global communication and memory optimizing transformations for low power signal processing systems ». Dans *The IEEE Workshop on VLSI Signal Processing* (1994), pp. 178–187.

[21] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. Phd thesis, University of Rennes I, 93.

[22] A. P. Chandrakasan, M. P. Potkonjak, R. Mehra, J. Rabaey et R. W. Brodersen. « Optimizing power using transformations ». *IEEE Transactions on CAD of Integrated Circuits and Systems* **14**, numéro 1 (1995), 12–30.

[23] A. Darte. « On the Complexity of loop Fusion ». *Parallel Computing* **26**, numéro 9 (2000), 1175–1193.

[24] A. Darte et G. Huard. « Loop shifting for loop compaction ». *International Journal of Parallel Programming* **28**, numéro 5 (2000), 499–534.

[25] A. Darte et G. Huard. « New Results on Array Contraction ». Dans *The IEEE International Conference On Application-Specific Sytems, Architectures, and Processors* (San Jose, California, 2002), pp. 359–370.

[26] C. Ding et K. Kennedy. « Memory Bandwidth Bottleneck and its Amelioration by a Compiler ». Dans *Proceedings of International Parallel and Distribute Processing Symposium* (Cancun, Mexico, 2000), pp. 181–190.

[27] C. Eisenbeis, W. Jalby, D. Windheiser et F. Bodin. « A Strategy for Array Management in Local Memory ». *Journal of Mathematical Programming : Series A* **63**, numéro 3 (1994), 331–370.

[28] P. Feautrier. « Some efficient solutions to the affine scheduling problem, part I : one-dimensional time ». *International Journal of Parallel Programming* **21**, numéro 5 (Octobre 1992), 313–348.

[29] P. Feautrier. « Some efficient solutions to the affine scheduling problem, part II : multi-dimensional time ». *International Journal of Parallel Programming* **21**, numéro 6 (Décembre 1992), 389–420.

[30] P. Feautrier et V. Lefebvre. « Storage management in parallel programs ». Dans *5th Euromicro Workshop on Parallel and Distributed Processing* (London, 1997), pp. 181–188.

[31] D. Gannon, W. Jalby et K. Gallivan. « Strategies for cache and local memory management by global program transformation ». *Journal of Parallel and Distributed Computing* **5**, numéro 10 (1988), 587–616.

[32] G. Gao, R. Olsen, V. Sarkar et R. Thekkath. « Collective Loop Fusion for Array Contraction ». Dans *Fifth Workshop on Languages and Compilers for Parallel Computing* (New Haven, CT, 1992).

[33] C. H. Gebotys. « Low energy memory and register allocation using network flow ». Dans *Design Automation Conference* (Anaheim, California, United States, 1997), pp. 435–440.

[34] E. Greef, F. Catthoor et H. Man. « Reducing storage size for static control programs mapped onto parallel architectures ». Dans *presented at Dagstuhl Seminar on Loop Parallelisation* (Schloss Dagstuhl, Germany, April 1996).

[35] E. Greef, F.Catthoor et H. Man. « Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications ». Dans *International Parallel Processing Symposium(IPPS) in Proceedings of the Workshop on Parallel Processing and Multimedia* (Geneva, Switzerland, April 1997), pp. 84–98.

[36] E. Greef, F.Catthoor et H. Man. « Array Placement for Storage Size Reduction in Embedded Multimedia Systems ». Dans *Proceedings of the International Conference on Application-Specific Array Processors (ASAP)* (Zurich, Switzerland, July 1997), pp. 66–75.

[37] C. Guettier. *Optimisation globale et placement d'applications de traitement du signal sur architectures parallèles en utilisant la programmation par contraintes*. Phd thesis, École Nationale Supérieure des Mines de Paris, 1997.

[38] G. Huard. *Algorithmique du décalage d'instructions*. Thèse de doctorat, Ècole Normale Supérieure de Lyon, 2001.

[39] F. Irigoin et R. Triolet. « Supernode partitioning ». Dans *Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA, 1988), pp. 319–329.

[40] M. Jiménez. *Multilevel Tiling for Non-Rectangular Iteration Spaces*. Phd thesis, Universitat Politècnica Catalunia Spain, 1999.

[41] M. Jiménez et al. « A general algorithm for tiling the register level ». Dans *Proceeding of the 12th ACM International Conference on Supercomputing* (Melbourne, Australia, 1998), pp. 133–140.

[42] M. Kamble et K. Ghose. « Analytical energy dissipation models for low power caches ». Dans *The international Symposium on Low Power Electronics and Design* (Monterey, California, United States, 1997), pp. 143–148.

[43] M. Kandemir, A. Choudhary et J. Ramanujam. « I/O-Conscious tiling for disk-resident data sets ». Dans *Euro-par99* (Toulouse, 1999), pp. 430–439.

[44] M. Kandemir, N. Vijaykrishnan, M. J. Irwin et H. S. Kim. « Experimental evaluation of energy behavior of iteration space tiling ». Dans *LCPC 2000* (Yorktown Heights, NY, USA, 2000), pp. 142–157.

[45] K.Danckaert, F.Catthoor et H. Man. « A loop transformation approach for combined parallelization and data transfer and storage optimization ». Dans *Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications, PDPTA'00* (Las Vegas NV, June 2000), pp. 2591–2597.

[46] K.Danckaert, F.Catthoor et H. Man. « A preprocessing step for global loop transformations for data transfer and storage optimization ». Dans *Proc. Intnl. Conf. on Compilers, Arch. and Synth. for Emb. Sys* (San Jose CA, Nov. 2000), pp. 34–40.

[47] W. Kelly et W. Pugh. « A framework for unifying reordering transformations ». *Technical Report CS-TR-3193, University of Maryland College Park,* (Apr. 1993).

[48] W. Kelly, W. Pugh et E. Rosser. « Code generation for multiple mappings ». Dans *The 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, Virginia, 1995), pp. 332–341.

[49] K. Kennedy. « Fast Greedy Weighted Fusion ». *International Journal of Parallel Programming* **29**, numéro 5 (2001), 463–491.

[50] K. Kennedy et K. S. McKinley. « Optimizing for parallelism and data locality ». Dans *Proceedings of the 1992 ACM International Conference On Supercomputing* (Washington, DC, USA, 1992), pp. 323–334.

[51] I. Kodukula, N. Ahmed et K. Pingali. « Data-centric multi-level blocking ». Dans *Proceedings of the ACM SIGPLAN 97 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, 1997), pp. 346–357.

[52] L. Lamport. « The Parallel Execution of DO Loops ». *Communications of the ACM* **17**, numéro 2 (Février 1974), 83–93.

[53] C. Lengauer et al. « Code generation in the polytope model ». Dans *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Paris, France, 1998), pp. 106–111.

[54] A. W. Lim, S. Liao et M. Lam. « Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning ». Dans *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Snowbird, Utah, USA,, 2001), pp. 103–112.

[55] N. Manjikian et T. S. Abdelrahman. « Fusion of loops for parallelism and locality ». *IEEE Transactions on Parallel and Distributed Systems* **8**, numéro 2 (1997), 193–209.

[56] K. McKinley, S. Carr et C. Tseng. « Improving data locality with loop transformations ». *ACM Transactions on Programming Languages and Systems* **18**, numéro 4 (1996), 424–453.

[57] K. S. McKinley et K. Kennedy. « Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution ». Dans *Languages and Compilers for Parallel Computing, 6th International Workshop* (Portland, Oregon, USA, 1993), pp. 301–320.

[58] N. Megiddo et V. Sarkar. « Optimal Weighted Loop Fusion for parallel Programs ». Dans *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, RI, USA, 1997), pp. 282–291.

[59] N. Museux. *Aide au placement d'applications de traitement du signal sur machines parallèles multi-SPMD.* Phd thesis, École Nationale Supérieure des Mines de Paris, 2001.

[60] T. J. F. Omnès. *Acropolis - un précompilateur de spécification pour l'exploration du transfert et du stockage des données en conception de systèmes embarqués à Haut débit.* Phd thesis, École Nationale Supérieure des Mines de Paris, 2001.

[61] M. Pedram. « Power minimization in IC design : Principles and applications ». *ACM Transactions on Design Automation of Electronic Systems* **1**, numéro 1 (1996), 3–56.

[62] W. Pugh et E. Rosser. « Iteration space slicing for locality ». Dans *Languages and Compilers for Parallel Computing, 12th International Workshop* (La Jolla/San Diego, CA, USA, 1999), pp. 164–184.

[63] F. Quilleré, S. Rajopadhye et D. Wilde. « Generation of efficient nested loops from polyhedra ». *International journal of parallel programming* **28**, numéro 5 (2000), 469–498.

[64] Équipe PIPS. « PIPS (Interprocedural Parallelizer for Scientific Programs) ». Dans *http ://www.cri.ensmp.fr.*

[65] V. Sarkar. « Automatic selection of high order transformations in the IBM XL FORTRAN compilers ». *IBM Journal of Research and Development* **41**, numéro 3 (1997), 233–264.

[66] V. Sarkar. « Optimized Unrolling of Nested Loops ». Dans *Proceedings of the 14th international conference on Supercomputing* (Santa Fe, New Mexico, United States, 2000), pp. 153–166.

[67] V. SarKar et G. Gao. « Optimizing of array accesses by collective loop transformations ». Dans *Proceedings of the 5th International Conference on Supercomputing* (Cologne, Germany, 1991), pp. 194–205.

[68] A. Schrijver. *Theory of linear and integer programming.* Wiley, NY, 1986.

[69] G. A. Silber. *Parallélisation automatique par insertion de directives.* Thèse de doctorat, Ècole Normale Supérieure de Lyon, 1999.

[70] P. Tang et J. Xue. « Generating efficient tiled code for distributed memory machines ». *Prallel Computing* **26**, numéro 1 (2000), 1369–1410.

[71] V. Tiwari, S. Malik et A. Wolfe. « Compilation Techniques for Low Energy : An Overview ». Dans *IEEE Symposium on Low Power Electronics* (San Diego, CA, 1994), pp. 38–94.

[72] M. van Swaaij, F. Franssen, F. Catthoor et H. D. Man. « Automating high-level control flow transformations for DSP memory management ». Dans *Proceedings of the IEEE workshop on VLSI signal processing* (Napa Valley, CA, October 1992).

[73] I. Verbauwhede, F. Catthoor, J. Vandewalle et H. D. Man. « Background memory management for the synthesis of algebraic algorithms on multi-processor dsp chips ». Dans *Proc.VLSI'89, Int. Conf. on VLSI,* (Munich, W.-Germany, August 1989), pp. 209–218.

[74] S. Wetzel. *Automatic code generation in the polytope model.* Phd thesis, University of Passau, 1993.

[75] M. E. Wolf. *Improving locality and parallelism in nested loops.* Phd thesis, University of stanford, 1992.

[76] M. E. Wolf et M. Lam. « A loop transformation theory and an algorithm to maximize parallelism ». *IEEE Transactions on Parallel and Distributed Systems* **2**, numéro 4 (1991), 452–471.

[77] M. E. Wolf, D. Maydan et Ding-Kai-Chen. « Combining loop transformations considering caches and scheduling ». *International Journal of Parallel Programming* **26**, numéro 4 (1998), 479–503.

[78] M. Wolfe. « Iteration space tiling for memory hierarchies ». *Parallel Processing for Scientific Computing* (1987), 357–361.

[79] M. Wolfe. « Optimizing Supercompilers for Supercomputers. ». *MIT Press, Cambridge MA,* (1989).

[80] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[81] S. Wuytack, F. Catthoor, L. Nachtergaele et H. D. Man. « Power Exploration for Data Dominated Video Applications ». Dans *Proceedings of the IEEE International Symposium on Low Power Electronics and Design, ISLPD* (Monterey, CA, August 1996), pp. 359–364,.

[82] S. Wuytack, J. Diguet, F. Catthoor et H. D. Man. « Formalized methodology for data reuse exploration for low-power hierarchical memory mappings ». *IEEE Transactions on VLSI Systems, special issue ISLPED'97* **6**, numéro 4 (December 1998), 529–537.

[83] J. Xue. « On tiling as a loop transformation ». *Parallel Processing Letters* **7**, numéro 4 (1997), 409–424.

[84] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, 2000.

[85] J. Xue et C.-H. Huang. « Reuse-driven tiling for improving data locality ». *International Journal of Parallel Programming* **26**, numéro 6 (1998), 671–696.

[86] W. Ye, N. Vijaykrishnan, M. Kandemir et M. J. Irwin. « The design and use of SimplePower : A cycle-accurate energy estimation tool. ». Dans *Proceedings of the 37th Conference on Design Automation* (Los Angeles, CA, USA, 2000), pp. 340–345.

[87] H. P. Zima et B. M. Chapman. *Supercompilers for parallel and vector computers*, volume 1. Addison-Wesley, 1990.

Optimisation des transferts de données pour le traitement du signal

Résumé

L'objectif de ces travaux est la réduction de l'énergie électrique utilisée durant l'exécution des programmes de traitement du signal dans les systèmes embarqués. Nous étudions plus particulièrement la réduction des transferts de données qui joue un rôle important dans cette consommation. Nous étendons les travaux précédents sur la fusion, le pavage, les transformations unimodulaires et la réutilisation de la mémoire.

Les principaux apports sont (1) l'extension de l'application du pavage à des séquences de nids de boucles, (2) l'introduction d'une nouvelle technique d'allocation de tampons dans le cas de la fusion sans et avec pavage, (3) le développement d'un pavage hiérarchique à deux niveaux et (4) l'extension de l'application d'une transformation unimodulaire à des nids de boucles non parfaits. D'abord, nous avons appliqué la fusion sur une séquence de nids de boucles qui n'ont pas nécessairement le même domaine d'itération et nous avons remplacé dans le code fusionné tous les tableaux temporaires par des tampons circulaires et séquentiels. Nous avons ensuite appliqué simultanément la fusion avec pavage sur une séquence de nids de boucles et nous avons introduit une nouvelle technique de réallocation des tableaux temporaires. Dans le dernier apport de cette thèse, nous avons développé une nouvelle technique qui consiste à appliquer une transformation unimodulaire sur un nid de boucles non parfait et ensuite à éliminer toutes les gardes ajoutées.

Une implantation complète de toutes les techniques proposées dans cette thèse a été effectuée dans *PIPS*, compilateur de recherche du CRI. Nos expériences montrent que les différentes techniques proposées dans cette thèse permettent de réduire considérablement le nombre de défauts de cache qui est notre principal objectif.

Mots clés : compilation, localité de données, transformation de code, énergie électrique.

Data transfer optimization for signal processing applications

Abstract

The goal of this work is to reduce the electrical energy used during the execution of the signal processing applications in embedded systems. We have studied data transfers which use about two third of the total energy. We have extended the previous works on the fusion, the tiling, the unimodular transformations and the memory re-use in order to reduce these transfers by improving locality.

The main contributions are (1) the extension of tiling to sequences of nested loops, (2) the introduction of a new technique for buffers allocation in the case of fusion with and without tiling (3) the development of a hierarchical tiling at two levels and (4) the extension of the unimodular transformation application to imperfectly nested loops. At first, we have applied the fusion to a sequence of nested loops which do not necessarily have the same iteration domain and we have replaced all the temporary arrays by circular and sequential buffers. Then we have applied simultaneously the fusion with the tiling to a sequence of nested loops and we have introduced a new technique for reallocation of the temporary arrays. The last contribution of this thesis, is a new technique which consists in applying an unimodular transformation on an imperfectly loop nest and of eliminating all the added guards.

All the techniques presented in this thesis are completely implemented in *PIPS*, the research compiler CRI. Our experiments have shown that the various techniques proposed in this thesis reduce the number of cache misses by a factor 5 for a cavity detection application.

Keywords : compilation, data locality, code transformation, electrical energy.