Elimination des gardes dans les transformées unimodulaires de boucles mal imbriquées

Youcef BOUCHEBABA
CRI/ ENSMP
77305 Fontainebleau Cedex
France
e-mail: { boucheba@cri.ensmp.fr}

16 mars 2001

Document EMP-CRI A/312

Résumé

Les optimisations de code à base de transformations unimodulaires ne peuvent s'appliquer qu'à des nids de boucles parfaitement imbriquées. 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, cet article propose une technique d'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées et non paramétriques à deux dimensions. On peut diviser ce travail en deux parties : dans la première, on s'intéresse à la transformation d'un nid de boucles mal imbriquées en un nid parfaitement imbriqué en ajoutant des gardes ad'hoc sur les instructions que l'on transfère dans la boucle la plus interne. Nous donnons la condition nécessaire et suffisante à l'application de cette transformation. Si cette condition n'est pas vérifiée, ou bien la deuxième boucle n'est jamais exécutée et dans ce cas on l'élimine entièrement, ou bien, on décompose le nid de boucles en une séquence de deux nids, tel que: dans un nid on fait entrer toutes les instructions dans la boucle la plus interne et dans l'autre on élimine la boucle interne. Dans la deuxième partie, on s'intéresse au problème inverse, l'élimination des gardes dans le code obtenu après l'application d'une transformation unimodulaire. On décompose ce dernier 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 et pouvoir sortir les gardes et les instructions gardées de la boucle interne, voir à éliminer complètement les gardes.

1 Introduction

Plusieurs travaux [14, 2, 8, 12] 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 leurs traitements pourraient permettre l'obtention de gain considérable. C'est pour ces raisons qu'on s'est intéressé 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. On est donc ramené à rendre ce dernier parfaitement imbriqué en ajoutant des gardes sur les instructions qu'on fait rentrer dans la boucle la plus interne avant de pouvoir lui appliquer une transformation unimodulaire. Wolf[11] 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 [13] 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 [15] a proposé un algorithme de génération de code sans gardes qui est différent de notre algorithme.

Dans cet article, on s'est intéressé à la condition nécessaire et suffisante pour

```
Begin
    do ( i = 0, i = 4 )
        S = 0
        do (j = 0 , j = 6 )
            S = S + A(i,j) * X(j)
        enddo
        Y(i) = S
    enddo
End
          (a)
```

```
Begin
    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
End
          (b)
```

FIG. 1 – *Produit matrice/vecteur: passage du nid mal imbriqué au nid parfaitement imbriqué.*

que la transformation d'un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées soit correcte. On a aussi traité le cas où cette condition n'est pas vérifiée. En deuxième lieu, on donne un algorithme qui permet d'éliminer les gardes dans le code transformé en découpant ce dernier en une séquence de nids de boucles. Chamski [4] 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 [5] et Wetzel [10] se sont intéressés aux techniques de fusion et de distribution de boucles pour la génération de code.

Cet article est organisé de la manière suivante. On commence par décrire le problème avec un simple exemple, le produit matrice/vecteur. Puis on donne la condition nécessaire et suffisante pour transformer un nid de boucles mal imbriquées en un nid de boucles parfaitement imbriquées. Ensuite on décrit l'algorithme qui permet d'éliminer les gardes dans le nid transformé. Ces deux algorithmes seront ensuite comparés aux travaux antérieures. Enfin, on termine par une conclusion récapitulant les résultats et présentant les travaux futurs.

2 Exemple introductif

Afin de mieux comprendre le problème traité, on donne un simple exemple qui est le produit matrice/vecteur, dont l'algorithme est présenté en figure 1:(a).

Pour appliquer une transformation unimodulaire sur l'algorithme (a) on doit 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`. Ceci est possible car la condition nécessaire et suffisante est vérifiée (`Section3`): pour toute valeur de i, il existe au moins une itération j . Le nid de boucles parfaitement imbriquées correspondant à l'algorithme (a) est donné en figure 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 de l'algorithme (b) par $T = \begin{pmatrix} 6 & 1 \\ 1 & 0 \end{pmatrix}$, nous donne l'algorithme de la figure 2 : (c).

```
Begin
    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
End
                (c)
```

FIG. 2 – *Algorithme transformé*

On voit que l'image de la droite définie par $j = 0$ qui est à la frontière du polytope définissant le domaine d'itérations de l'algorithme (b) est la droite définie par $-i' + 6 * j' = 0$ qui est à la frontière du polytope du domaine de l'algorithme (c).

Lorsqu'on transforme un nid mal imbriqué en un nid parfaitement imbriqué, on ajoute des gardes sur les instructions qu'on fait 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, on doit décomposer le nid de boucles (c) 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 (c) en une séquence de la figure 4:

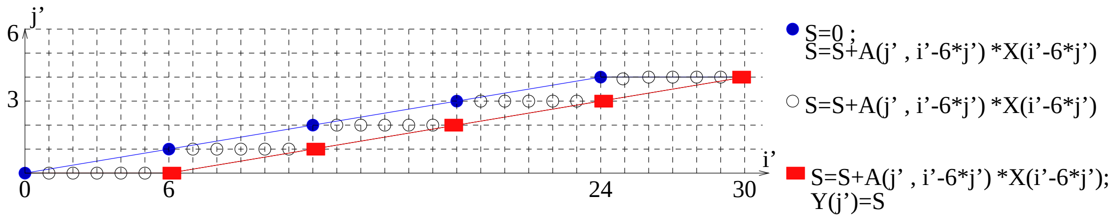

FIG. 3 – *Domaine d'itérations de l'algorithme (c)*

L'étude du code transformé en figure 2 et du domaine d'itérations en figure 3 montre que l'on peut 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'exécutant 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.

Begin

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

End

FIG. 4 – *Code transformé et décomposé*

```
Begin
    do (i1 = inf, i1 = sup)
       S1
       do (i2 = a1*i1+b1 , i2 = a2*i1+b2 )
          S2
       enddo
       S'1
    enddo
End
       (d)
```

```
Begin
    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 S'1
       enddo
    enddo
End
       (e)
```

FIG. 5 – *Programmes traités*

3 Transformation en nid de boucles parfaitement imbriquées

Les nids de boucles qu'on traite dans cet article sont de la forme (d), avant de leurs appliquer une transformation unimodulaire on doit les mettre sous la forme (e). Ceci est possible si et seulement si la $proposition2$ est vérifiée et il ne y a ni dépendances de données ni de contrôles entre S_1 (ou S_1') et la boucle i_2. Avant de démontrer la $proposition2$, nous introduisons la $proposition1$ suivante:

Remarque

S'il y a des dépendances de données ou de contrôles entre S_1 (ou S_2) et la boucle i_2 on peut résoudre ce problème par le renomage de i_2.

Proposition1

Soit inf et sup deux entiers tels que $inf \leq sup$ alors on a l'équivalence suivante:

$$\left\{\begin{array}{l} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{array}\right. \iff \left\{\begin{array}{l} (a_1 = a_2) \wedge (b_2 \geq b_1) \\ \quad\quad \vee \\ (a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \\ \quad\quad \vee \\ (a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1) \end{array}\right.$$

Preuve:

1) $\Longrightarrow$?

On suppose que les deux conditions suivantes $\left\{\begin{array}{l} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{array}\right.$ sont vérifiées.

On distingue les 3 cas $a_1 = a_2$, $a_1 > a_2$, $a_1 < a_2$.

Si $a_1 = a_2$ alors nécessairement $b_2 \geq b_1$.

Si $a_1 > a_2$ on a par supposition $(a_2 * sup + b_2 \geq a_1 * sup + b_1)$.

Si $a_1 < a_2$ on a aussi par supposition $(a_2 * inf + b_2 \geq a_1 * inf + b_1)$.

2) $\Longleftarrow$?

Si $(a_1 = a_2) \wedge (b_2 \geq b_1) \Longrightarrow \begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases}$

Si $(a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \Longrightarrow (a_1 - a_2) * sup \leq b_2 - b_1$ comme $(a_1 - a_2)$ est positif et $sup > inf$ alors $(a_1 - a_2) * inf \leq b_2 - b_1 \Longrightarrow a_2 * inf + b_2 \geq a_1 * inf + b_1$.

Si $(a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1)$ alors $(a_2 - a_1) * inf \geq b_1 - b_2$ comme $(a_2 - a_1)$ est positif et $inf \leq sup$ alors $(a_2 - a_1) * sup \geq b_1 - b_2 \Longrightarrow (a_2 * sup + b_2 \geq a_1 * sup + b_1)$.

Fin preuve
Proposition2:

Le programme (d) et le programme (e) 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 programme (d) et (e) 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 faite 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
$\begin{cases} inf \leq i_1 \leq sup \\ a_1 * i_1 + b_1 > a_2 * i_1 + b_2 \end{cases} = \begin{cases} inf \leq i_1 \leq sup \\ \\ (a_1 - a_2) * i_1 > b_2 - b_1 \end{cases} \quad (I)$ est vide.

Cas1 $a_1 = a_2$**:** Le domaine définie par (I) n'admet aucune solution si est seulement si $b_2 \geq b_1$.

Cas2 $a_1 > a_2$**:** Le domaine définie par (I) n'admet aucune solution $\Longleftrightarrow \frac{b_2 - b_1}{a_1 - a_2} \geq sup$ c'est à dire $a_2 * sup + b_2 \geq a_1 * sup + a_1$.

Cas3 $a_1 < a_2$**:** le domaine définie 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 programme (d) est équivalent au programme $(e) \Longrightarrow$

$$\begin{cases} (a_1 = a_2) \wedge (b_2 \geq b_1) \\ \quad\quad \vee \\ (a_1 > a_2) \wedge (a_2 * sup + b_2 \geq a_1 * sup + b_1) \quad (II) \\ \quad\quad \vee \\ (a_1 < a_2) \wedge (a_2 * inf + b_2 \geq a_1 * inf + b_1) \end{cases}$$

D'après la *proposition*1 et (II) le programme (d) est équivalent au programme (e) $\Longrightarrow$

$$\begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases}$$

2) $\Longleftarrow$**?**

$$\begin{cases} a_2 * sup + b_2 \geq a_1 * sup + b_1 \\ a_2 * inf + b_2 \geq a_1 * inf + b_1 \end{cases} \Longrightarrow \forall i_1 \in [inf,sup] : a_1 * i_1 + b_1 \leq a_2 * i_1 + b_2$$

car les fonctions sont monotones: $\Longrightarrow$

$$\begin{cases} \forall i_1 \in [inf,sup], \exists!\ i_2 = a_1 * i_1 + b_1 /\ e(S_1(i_1,i_2))\ sera\ executee \\ \forall i_1 \in [inf,sup], \exists!\ i_2 = a_2 * i_1 + b_2 / e(S_1'(i_1,i_2))\ sera\ executee \end{cases}$$

Donc le nombre d'occurences de S_1 dans le programme (d) et le programme (e) sont égaux, de même pour S_1'. Et on peut vérifier facilement que l'ordre d'exécution des deux programmes et le même. Par conséquent le programme (d) est (e) sont équivalents.

Fin preuve

3.1 Décomposition en séquence de nids de boucles

D'après la *proposition*2, le programme (d) peut être mis sous 4 formes, suivant la position des droites définies par $i_2 = a_1 * i_1 + b_1$ et $i_2 = a_2 * i_1 + b_2$. On pose $f_1(i_1) = a_1 * i_1 + b_1$ et $f_2(i_2) = a_2 * i_1 + b_2$.

Cas1: $f_2(inf) \geq f_1(inf) \wedge f_2(sup) \geq f_1(sup)$
Le programme (d) sera mis sous la forme suivante:

Begin
    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
End

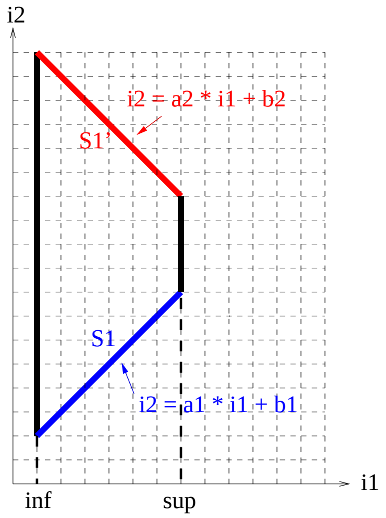

FIG. 6 – *cas1*

La figure 6 illustre un exemple qui entre dans le cas 1 où $f_1(i_1) = i_1 + 1$ et $f_2(i_1) = -i_1 + 19$.

Cas2: $f_2(inf) \geq f_1(inf) \wedge f_2(sup) < f_1(sup)$

Soit e un réel tel que $f_1(e) = f_2(e)$, alors le programme (d) sera mis sous la forme suivante:

Begin

```
    do (i1 = inf, i1 = ⌊e⌋)
       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
    do (i1 = ⌊e⌋+1 , i1 = sup)
       S1
       S1'
    enddo
End
```

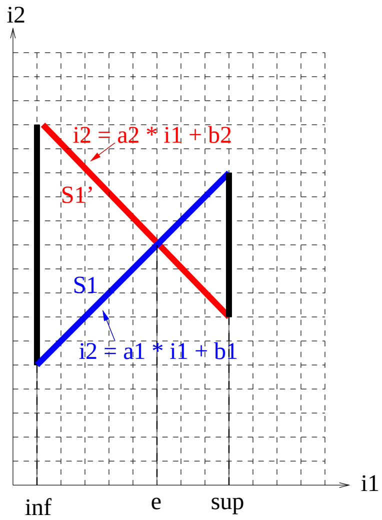

FIG. 7 – *cas2*

La figure 7 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.

Cas3: $f_2(inf) < f_1(inf) \wedge f_2(sup) \geq f_1(sup)$

Soit e un réel tel que $f_1(e) = f_2(e)$, alors le programme (d) sera mis sous la forme suivante:

Begin
```
    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
End
```

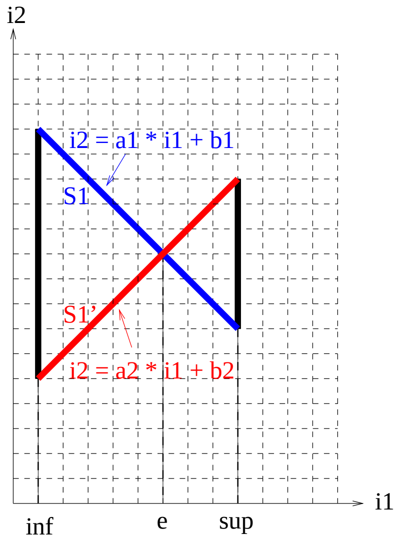

FIG. 8 – *cas3*

La figure 8 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.

Cas4: $f_2(inf) < f_1(inf) \wedge f_2(sup) < f_1(sup)$

Le programme (d) sera mis sous la forme suivante:

Begin
```
    do (i1 = inf, i1 = sup)
        S1
        S1'
    enddo
End
```

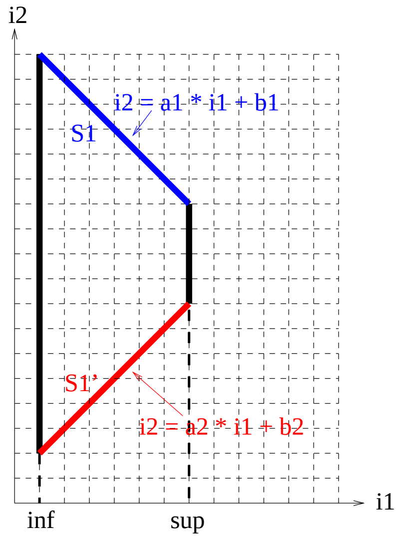

FIG. 9 – *cas4*

La figure 9 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.

Remarque

La transformation T sera appliquée uniquement au bloc de programme constitué de deux boucles.

4 Elimination des gardes dans le nid de boucle transformé

Cette section présente les propositions et 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.

Le nid de boucles aura la forme de la figure 10 après l'application de la transformation unimodulaire T.

Begin
```
    do (-i'_1 ≤ -b_0, i'_1 ≤ b'_0)
        do (∧_{j=1}^{n}(a_j * i'_1 - d_j * i'_2 ≤ b_j), ∧_{j=1}^{n'}(a'_j * i'_1 + d'_j * i'_2 ≤ 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
End
```

(f)

FIG. 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 nid (f) mais 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 programme (f) 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.

On veut ramener le programme (f) sous la forme de la figure suivante:

```
Begin
    do (−i'_1 ≤ −b_0, i_1 ≤ b'_0)
        S_1
        do (∧_{j=1}^{n}(a_j * i'_1 − d_j * i'_2 ≤ b_j), ∧_{j=1}^{n'}(a'_j * i'_1 + d'_j * i'_2 ≤ b'_j ))
            S_2
        enddo
        S'_1
    enddo
End
```

(g)

FIG. 11 – *Programme sans gardes*

Proposition3:

Les programmes (f) et (g) sont équivalents si et seulement si, les conditions suivantes sont vérifiées:

$$\begin{cases} n = n' = 1 \\ (h_{m,1} = a_j) \wedge (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \\ (h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \end{cases}$$

Preuve:

1)$\Longrightarrow$?

Les programmes (f) et (g) sont équivalents alors:

- 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* $h_{m,2} = -1$.
- 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:
  Cas1: $-h_{m,2} = d_j$ *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* $h_{m,1} = a_j$

  Cas2: $h_{m,2} = d_j$ *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* $h_{m,1} = -a_j$

Cas1 et Cas2 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 dj=1.

En résumé, les programmes (f) et (g) 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 programmes (f) et (g) sont équivalents alors:

$$\begin{cases} n' = 1 \\ (h'_{m,1} = a'_j) \wedge (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \end{cases}$$

2)$\Longleftarrow$?

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 programme (f) va se mettre sous la forme suivante:

```
Begin
    do (−i'1 ≤ −b0, i'1 ≤ b'0)
        do (a1 ∗ i'1 − i'2 ≤ b1, a'1 ∗ i'1 + i'2 ≤ b'1 )
            If (a1 ∗ i'1 − i'2 = b1) alors S1
            S2                        (d)
            If (a'1 ∗ i'1 + i'2 = b'1) alors S'1
        enddo
    enddo
End
```

FIG. 12 – *Programme avec gardes*

et le programme (g) va se mettre sous la forme donnée en figure 13.

```
Begin
    do (−i'1 ≤ −b0, i'1 ≤ b'0)
        S1
        do (a1 ∗ i'1 − i'2 ≤ b1, a'1 ∗ i'1 + i'2 ≤ b'1 )
            S2                        (c)
        enddo
        S'1
    enddo
End
```

FIG. 13 – *Programme sans gardes*

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 proposition2 le programme (f) est équivalent au programme (g).

Fin preuve

4.1 Décomposition minimale du nid transformé

Malheureusement n et n' ne sont pas toujours égaux à 1 par conséquent on doit 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 ensembles suivants:
- $H_1^- = \{-i'_1 \leq -b_0\}$.
- $H_1^+ = \{i'_1 \leq b'_0\}$.
- $H_2^- = \{a_j * i'_1 - d_j * i'_2 \leq b_j,\ j = 1 : n\}$.
- $H_2^+ = \{a'_j * i'_1 + d'_j * i'_2 \leq b'_j,\ j = 1 : n'\}$.

Sachant que d_j et d'_j sont des entiers positifs non nuls.

On note:
- $C^- = \{c_{l,k}(i_1) = \frac{a_j * i'_1 - b_j + d_j - 1}{d_j}\ /\ (a_j * i'_1 - d_j * i'_2 \leq b_j) \in H_2^-\}$.
- $C^+ = \{c'_{l,k}(i_1) = \frac{-a'_j * i'_1 + b'_j}{d'_j}\ /\ (a'_j * i'_1 + d'_j * i'_2 \leq b'_j) \in H_2^+\}$.
- $\vec{j} = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$ le vecteur d'itération initiale.
- $\vec{j}' = \begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix}$ le vecteur d'itération après la transformation T.
- $(h_{m,1} * i'_1 + h_{m,2} * i'_2 = b_m)$ est équivalent à $(h_m\ \vec{j}' = b_m)$.
- $(h'_{m,1} * i'_1 + h'_{m,2} * i'_2 = b_{m'})$ est équivalent à $(h'_m\ \vec{j}' = b'_m)$.

Proposition4

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 $\begin{cases} a_1 = ig_1 \\ b_p = ig_p \\ (b_i = ig_i \wedge a_i = ig_i + 1) ou (b_i = ig_i - 1 \wedge a_i = ig_i) pour\ i = 2,p-1 \end{cases}$
est le découpage minimale 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 minimale?

On suppose qu'il existe un autre découpage qui contienne moins d'intervalles => il existe un intervalle [a,b] tels 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.

Fin preuve

4.2 Isolement d'un sommet générateur

Dans certains cas, illustrés en figures 14, 15 et 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 figure 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 figure 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 14(a) et 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 palier à 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 2 cas particuliers pour l'itération $i'_1 = ig_i$.

Dans les exemples 14(c) et 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 14(e) et 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 $S_1;\ S_2;\ S'_1$.

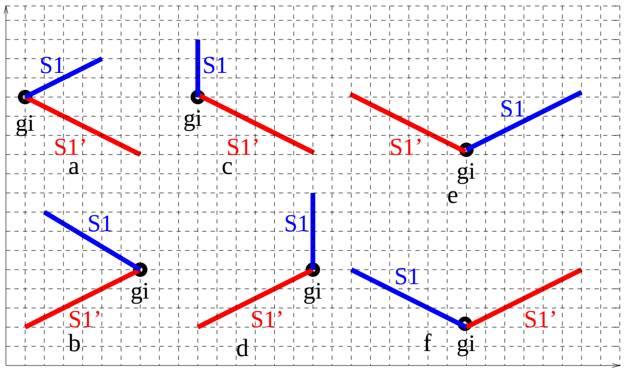

FIG. 14 – *Les différents cas de la condition1*

De manière symétrique, si on a un sommet générateur qui satisfait la condition 2 (voir figure 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.

Si on a un sommet générateur qui correspond à l'un des cas de la figure 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 (voir section 4.4).

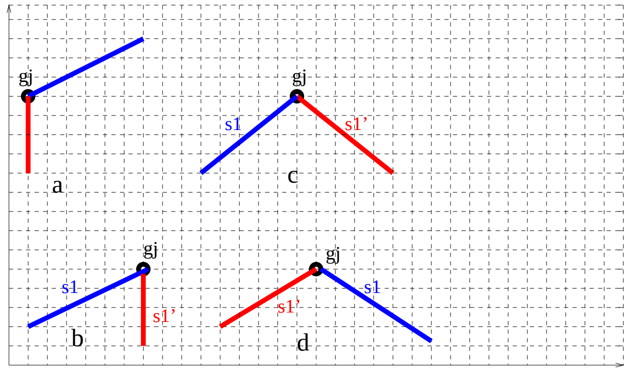

FIG. 15 – *Les différents cas de la condition2*

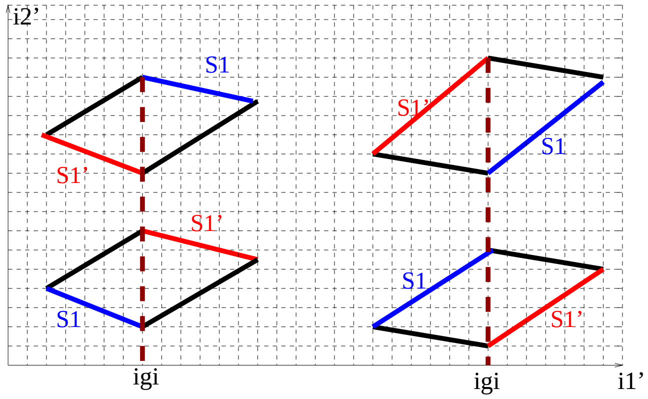

FIG. 16 – *Cas particulier*

4.3 Calcul des bornes de boucles et code généré

La proposition4 et le traitement particulier de certains sommets pour lesquels on génère un nid de boucles spécifique, conduisent au code intermédiaire suivant:

Begin

```
    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
End
```

Sachant que:

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.

4.4 Simplification d'un nid de boucles

Les conditions de la $proposition3$ ne sont pas toujours vérifiées et la $proposition4$ 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 proposition3 on peut sortir le code S_1 avant la deuxième boucle.
  2. $h_{m,2} \neq -1$, d'après la proposition3 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 proposition3 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 proposition3, on peut extraire le code S'_1 après la deuxième boucle.
  2. $h'_{m,2} \neq 1$, d'après la proposition3, 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 proposition3, 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 $If \ mod(b'_m - h'_{m,1} * i'_1 \ , h'_{m,2}) \ then \ S_2 \ ; \ S'_1$.

4.5 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.

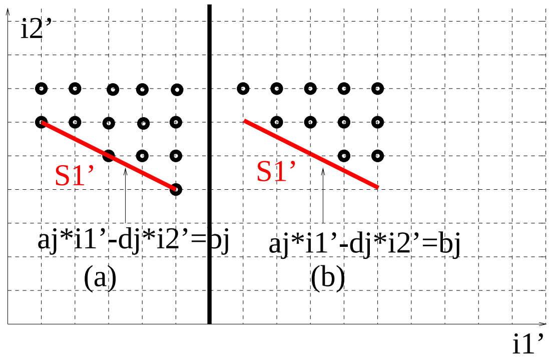

FIG. 17 – *Avant et après l'extraction de* S_1'

Si le demi-espace de la figure 17: (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 17: (b) qui est obtenu à partir du demi-espace de la figure 1.17 (a)en éliminant les points entiers sur le droite définie par $(a_j * i_1' - d_j * i_2 = b_j)$, sera caractérisé par $(i_2' \geq \frac{a_j * i_1' - b_j}{d_j} + 1)$.

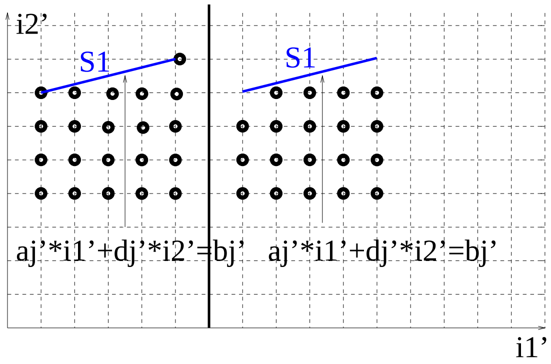

FIG. 18 – *Avant et après l'extraction de* S_1'

De même, si le demi-espace de la figure 18: (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 18: (b) qui est obtenu à partir du demi-espace de la figure 1.18(a) en éliminant les points entiers de la droite définie par $(a'_j * i'_1 + d'_j * i_2 = b'_j)$, sera caractérisé par $(i'_2 \leq \frac{-a'_j * i'_1 + b'_j}{d'_j} - \frac{1}{d_j})$.

5 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'extérieure 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.

5.1 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 suivant:

Begin

```
    do ( i'1 = ig1 , i'1 = ig1 )
        code1,1
        do ( i'2 = c1,1 , i'2 = c'1,1)
            α1
            S2
            β1
        enddo
        code'1,1
    enddo
    .
    .
    do ( i'1 = igj , i'1 = igj )
        codej,j
        do ( i'2 = cj,j , i'2 = c'j,j)
            S2
        enddo
        codee'j,j
    enddo
    do ( i'1 = igj + 1, i'1 = igj+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})
            α_p
            S_2
            β_p
        enddo
        code'_{p,p}
    enddo
End
```

5.2 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 éliminer dans la boucle la plus interne.

Begin

$$\alpha_1 = \begin{cases} \text{``}\ S1\ \text{''} & si\ \ (h_{m,0} = 1) \wedge (h_{m,1} = 0) \wedge (b_m = ig_1) \\ \text{``}\ ;\ \text{''} & sinon \end{cases}$$

$$\alpha_p = \begin{cases} \text{``}\ S1\ \text{''} & si\ \ (h_{m,0} = 1) \wedge (h_{m,1} = 0) \wedge (b_m = ig_p) \\ \text{``}\ ;\ \text{''} & sinon \end{cases}$$

$$\beta_1 = \begin{cases} \text{``}\ S1'\ \text{''} & si\ \ (h'_{m,0} = 1) \wedge (h'_{m,1} = 0) \wedge (b'_m = ig_1) \\ \text{``}\ ;\ \text{''} & sinon \end{cases}$$

$$\beta_p = \begin{cases} \text{``}\ S1'\ \text{``} & si\ \ (h'_{m,0} = 1) \wedge (h'_{m,1} = 0) \wedge (b'_m = ig_p) \\ \text{``}\ ;\ \text{''} & sinon \end{cases}$$

End

5.3 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 suivant :

```
Begin
    soit nbr le nombre de nids generes.
    si ∃ un sommet generateur  g_s qui satisfait condition_1  ou condition_2 alors
       soit nid_k le nid correspondant 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
End
```

$Condition_1()$ et $Condition_2()$ sont détaillées dans la section 4.2.

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 section 4.3.

Le fonctionnement de la procédure $Traiterspecial$ est similaire à celui de la procédure $Traiter$, sauf que le code extrait avant ou après la deuxième boucle est différent.

Procédure Traiter

Cette procédure calcule $code_{p,q}$, $garde'_{p,q}$, $c_{p,q}$ et $c'_{p,q}$.

```
Begin
```

$$code_{p,q} = \begin{cases} \text{“}S_1\text{''} & si\ (h_{m,1} = a_j)\wedge \\ & \quad (h_{m,2} = -d_j = -1) \wedge (b_m = b_j) \\ \text{“}If\ entier(\frac{b_m - h_{m,1} * i'_1}{h_{m,2}})\ then\ S_1\text{''} & si\ (h_{m,1} = a_j)\wedge \\ & \quad (h_{m,2} = -d_j \neq -1) \wedge (b_m = b_j) \\ \text{“}\ S_2\ ;\ S'_1\text{''} & si\ (h'_{m,1} = a_j)\wedge \\ & \quad (h'_{m,2} = -d_j = -1) \wedge (b'_m = b_j) \\ \text{“}\ If\ entier(\frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}})\ then\ S_2\ ;\ S'_1\text{''} & si\ (h'_{m,1} = a_j)\wedge \\ & \quad (h'_{m,2} = -d_j \neq -1) \wedge (b'_m = b_j) \\ \text{“}\ ;\ \text{''} & sinon \end{cases}$$

$$code'_{p,q} = \begin{cases} \text{“}\ S_1\ ;\ S_2\text{''} & si\ (h_{m,1} = a'_j)\wedge \\ & \quad (h_{m,2} = d'_j = 1) \wedge (b_m = b'_j) \\ \text{“}\ If\ entier(\frac{b_m - h_{m,1} i'_1}{h_{m,2}})\ then\ S_1\ ;\ S_2\text{''} & si\ (h_{m,1} = a'_j)\wedge \\ & \quad (h_{m,2} = d'_j \neq 1) \wedge (b_m = b'_j) \\ \text{“}\ S'_1\text{''} & si\ (h'_{m,1} = a'_j)\wedge \\ & \quad (h'_{m,2} = d'_j = 1) \wedge (b'_m = b'_j) \\ \text{“}\ If\ entier(\frac{b'_m - h'_{m,1} * i'_1}{h'_{m,2}})\ then\ S'_1\text{''} & si\ (h'_{m,1} = a'_j)\wedge \\ & \quad (h'_{m,2} = d'_j \neq 1) \wedge (b'_m = b'_j) \\ \text{“}\ ;\ \text{“} & sinon \end{cases}$$

$$c_{p,q} = \begin{cases} \frac{a_j * i_1 - b_j}{d_j} + 1 & si\ (h'_{m,1} = a_j) \wedge (h'_{m,2} = -d_j) \wedge (b'_m = b_j) \\ \frac{a_j * i_1 - b_j + d_j - 1}{d_j} & sinon \end{cases}$$

$$c'_{p,q} = \begin{cases} \frac{-a'_j * i'_1 + b'_j}{d'_j} - \frac{1}{d_j} & si\ (h_{m,1} = a'_j) \wedge (h_{m,2} = d'_j) \wedge (b_m = b'_j) \\ \frac{-a'_j i'_1 + b'_j}{d'_j} & sinon \end{cases}$$

```
End
```

Traiterspecial

Cette procédure traite le nid de boucles qui correspond au sommet générateur qui satisfait le condition1 ou la condition2.

Begin

$$code_{s,s} = \begin{cases} \text{“ } S_1 \ ; S_2 \ ; S_1' \text{ ''} & si \ \ condition_1(g_s) \\ \text{“ ; ''} & sinon \end{cases}$$

$$code'_{s,s} = \begin{cases} \text{“ } S_1 \ ; S_2 \ ; S_1' \text{ ''} & si \ \ condition_2(g_s) \\ \text{“ ; ''} & sinon \end{cases}$$

$$c_{s,s} = \begin{cases} \frac{a_j i_1 - b_j}{d_j} + 1 & si \ \ condition_1(g_s) \\ \frac{a_j i_1 - b_j + d_j - 1}{d_j} & sinon \end{cases}$$

$$c'_{s,s} = \begin{cases} \frac{-a'_j i'_1 + b'_j}{d'_j} - \frac{1}{d_j} & si \ \ condition_2(g_s) \\ \frac{-a'_j i'_1 + b'_j}{d'_j} & sinon \end{cases}$$

End

5.4 Etape 4

Pour tous les sommets générateurs qui ne sont pas dans l'un des cas de la section 4.2, on regroupe leur code avec le code de l'intervalle gauche ou droite qui leur sont adjacents.

6 Travaux antérieures

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 [14] 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 des dépendances cycliques.

```
{ 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;
   }
}
```

FIG. 19 – *Produit Matrice /Vecteur: Code généré par l'algorithme de Quilleré*

Wolf [11] 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 [15] 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 la 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 [4], Boulet [3], Quilleré [6] et Omega [7, 9].

Chamski [4] 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 [3] génère aussi un code qui scanne les polyèdres. le code généré est différent du nôtre car il ne contient pas des boucles "DO".

```
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
```

FIG. 20 – *Produit Matrice/Vecteur : Code généré par notre algorithme*

L'Omega [7, 9] et Quileré [6] 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 [7, 9] 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 21, il sort les codes gardés, à l'extérieure de la boucle la plus interne sans les gardes "if entier $(\frac{i'}{6})''$ et "if entier $(\frac{i'-6}{6})$".

Par contre, Quileré [6] 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-ensemnble 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 19 et 24.

```
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)
   }
}
```

FIG. 21 – *Produit Matrice/Vecteur: Code généré par l'Omega calculateur*

Notre travail est donc très proche de celui de Quilleré [6] et de Omega [7, 9] calculateur. Comme nous traitons un cas particulier, notre algorithme est plus rapide et plus facile à implémenter. Les figures 19, 20 et 21 montrent respectivement les codes générés par l'algorithme de Quilleré [6], par notre algorithme et par l'Omega calculateur [7, 9] pour l'exemple du produit matrice vecteur, après l'application de la transformation unimodulaire qui est decrite en section 2.

```
for (i = 0; i <= 5; i++) {
   for (j = i; j <= 5; j++) {
      if (i-j==0) S1 ;
      S2;
      if (j-5==0) S3;
   }
}
```

FIG. 22 – *Exemple générique: Code avant l'élimination des gardes*

```
for (i = 0; i <= 5; i++) {
   S1;
   for (j = i; j <= 5; j++) {
      S2;
   }
   S3;
}
```

FIG. 23 – *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;
   }
}
```

FIG. 24 – *Exemple générique: Code généré par l'algorithme de Quilleré*

On remarque sur les figures 19 et 21 que l'algorithme de Quilleré [6] isole les codes correspondant à i = 0 et i=30 sons que cela soit nécessaire et que l'Omega calculateur [7, 9] génère un code faux parce qu'il omet des gardes nécessaires pour les instructions Y(j)=S et S=0. En effet ces dernieres 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 23 et 24 montrent respectivement les codes générés par notre algorithme et par l'algorithme de Quilleré [6] pour l'exemple de la figure 22 après l'élimination des gardes. On voit que l'algorithme de Quilleré [6] isole inutilement certains sous-ensembles du domaine d'itérations sans que cela soit nécessaire.

7 Conclusion

L'élimination des gardes dans les transformées unimodulaires de boucles mal imbriquées a été traitée par peux d'auteur. Nous avons traité les deux problèmes en détail: (1) la mise sous formes parfaitement imbriqué 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.

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.

Références

[1] W. Abu-Sufah. Improving the performance of virtual memory computers. *PhD thesis, University of Illinois at Urbana-Champaign*, Nov. 1978.

[2] U. Banerjee. Loop transformations for restructuring compilers. *Kluwer academic publishers*, 93.

[3] P. Boulet and P. Feautrier. Scanning polyhedra without do-loops. *Parallel archetectures and compilations techniques*, Paris, 1998.

[4] Z. Chamski. Environnement logiciel de programmation d'un accélérateur de calcul parallèle. *PhD thesis, University of Rennes I*, 93.

[5] C. Lengauer et al. Code generation in the polytope model. *Pact98*, 98.

[6] S. Rajopadhye F. Quilleré and D. Wild. Generation of efficient nested loops from polyhedra.

[7] W. Kelly and W. Pugh. A framework for unifying reordering transformations. *Technical Report CS-TR-3193, University of Maryland College Park,*, Apr. 1993.

[8] A. Choudhary M. Kandemir and J. Ramanujam. I/o-conscious tiling for disk-resident data sets. *Euro-par*, 1999.

[9] W. Pugh W. Kelly and E. Rosser. Code generation for multiple mappings. *In frontiers 95: The 5th Symposium on the frontiers of massively parallel computation*, Mc Lean, VA, Feb. 1995.

[10] S. Wetzel. Automatic code generation in the polytope model. *PhD thesis, University of Passau*, 93.

[11] M. E. Wolf. Improving locality and parallelism in nested loops. *PhD thesis, University of stanford*, 1992.

[12] M. E. Wolf and M. S. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE trans. on paral*, 2, 91.

[13] M. Wolfe. Optimizing supercompilers for supercomputers. *MIT Press*, 1989.

[14] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley*, 1996.

[15] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel computing*, 22(12):1621–1621, 97.