INSTITUT DE LA FRANCOPHONIE
POUR L'INFORMATIQUE

ECOLE NATIONALE SUPERIEURE
DES MINES DE PARIS


**Mémoire de fin d'études**

# Maintenance et mise au point de programmes : UTILISATION DU GRAPHE DE DEPENDANCE ET DE L'INFORMATION SUR LES ALIAS

*Auteur* : PHAM Quoc Dat
Promotion VI

*Responsable* : Corinne Ancourt
François Irigoin

**Fontainebleau, août 2002**

# Remerciements

*Mon stage de fin d'études au Centre de Recherche en Informatique (CRI), de l'Ecole Nationale Supérieure des Mines de Paris (ENSMP) s'est terminé. Ce mémoire présente tout mon travail. Avant tout, je souhaite exprimer mes remerciements à ceux qui m'ont aidé à bien finir le stage.*

*Je voudrais d'abord remercier Mme Corinne Ancourt, mon responsable de stage. Elle a consacré son temps précieux pour m'aider à surmonter des problèmes difficiles survenus durant le stage. Je lui dois une profonde reconnaissance pour ses instructions détaillées et sa gentillesse.*

*Ensuite, j'adresse toute ma gratitude à M. François Irigoin qui m'a reçu dans son équipe de travail et m'a initié à la recherche dans le domaine des analyses et transformations de programme. Ses suggestions, ses critiques et ses conseils ont contribué à l'aboutissement de ce travail.*

*Je remercie également le Professeur Robert Mahl, directeur du Centre de Recherche en Informatique, pour m'avoir chaleureusement accueilli dans son laboratoire.*

*Monsieur Georges-André Silber a corrigé pour moi le rapport avec enthousiasme, je lui réserve ma reconnaissance profonde.*

*Enfin, j'aimerais bien remercier tout le personnel du CRI pour leur collaboration étroite.*

***Fontainebleau, août 2002.***

# Table de matières

# Introduction

De nombreuses applications scientifiques dans les domaines de la météorologie, de la défense, de l'énergie nucléaire et de la sismique manipulent un grand nombre de données. Pour rester compétitive et prendre en considération le nombre de données toujours croissant, ces applications peuvent être traitées en parallèle.

La parallélisation d'un programme séquentiel comporte plusieurs phases essentielles: l'analyse des dépendances, les transformations du programme et enfin la parallélisation. Je me concentre dans ce mémoire sur la première phase de parallélisation, c'est-à-dire l'analyse des dépendances. Pour la parallélisation, le programme doit être restructuré et transformé. Une transformation peut être appliquée au programme si toutes les dépendances entre les instructions du programme sont respectées. Le premier chapitre de ce mémoire introduit les généralités sur la parallélisation et le concept de dépendance.

Le résultat de la phase d'analyse des dépendances est le graphe de dépendance de données. Ce graphe joue toujours un rôle très important dans la parallélisation. Le deuxième chapitre de ce mémoire présente l'implémentation d'une phase manquante dans PIPS, le filtrage du graphe de dépendance sur certaines variables et la représentation du graphe de dépendance sous forme graphique.

Le troisième chapitre présente les effets et l'implémentation d'une phase manquante dans PIPS, le filtrage des effets sur certaines variables et la représentation du graphe de flot de contrôle avec les effets associés.

Le graphe de dépendance est calculé par rapport à une procédure, mais les appels de procédures qui peuvent provoquer des alias entre des variables formelles ne sont pas pris en compte. Ce problème devient très important parce qu'il peut influencer la sémantique du programme. Par conséquent, les transformations du programme basées sur l'ordonnancement peuvent ne plus être correctes. Le dernier chapitre est dédié à l'étude des impacts des alias sur le graphe de dépendance.

# Chapitre I. Etat de l'art

De nombreux modèles d'architecture parallèles (i.e. MIMD, SIMD, VLIW) ont été proposés pour exploiter le parallélisme de plusieurs applications à différents niveaux (i.e. tâche, programme, sous-routine, boucle, instruction). Je présente brièvement les types principaux d'architectures parallèles dans la première section de ce chapitre.

Le parallélisme potentiel des applications doit être exploité de manière à utiliser au mieux toutes les ressources parallèles de ces architectures. Ce parallélisme peut être spécifié explicitement par le programmeur à l'aide des primitives d'un langage parallèle. Le parallélisme peut également être détecté par les compilateurs/paralléliseurs. Dans la deuxième section de ce chapitre, je présente quelques langages de programmation parallèle.

Le calcul de dépendances est une des phases les plus importantes pour la détection et l'exploitation du parallélisme implicite des programmes. Je consacre la troisième section de ce chapitre à présenter la notion de dépendance de données.

Une présentation de PIPS et la représentation de dépendance dans PIPS sont indispensables pour amener les lecteurs au contexte de travail. Je les présente dans la quatrième section de ce chapitre.

La dernière section de ce chapitre va présenter l'outil *daVinci* d'affichage graphique que j'utilise pour visualiser les graphes dans mes implémentations.

## *1.1 Architectures parallèles*

Comme leur nom l'indique, les machines parallèles sont des ordinateurs pouvant exécuter plusieurs instructions simultanément. Elles possèdent plusieurs processeurs, un processeur avec plusieurs unités de traitement ou une unité de pipeline ou encore une combinaison des trois. Il faut noter que tous les microprocesseurs récents comme les super-scalaires (Super SPARC, RS6000, MC88100) et les super-pipelines (R4000) sont des machines parallèles. De nombreux modèles d'architecture parallèle ont été proposés. La classification de l'architecture des machines la plus connue est celle de Flynn [Flynn66] qui est basée sur les flux d'instructions et de données. Cette classification comporte quatre catégories: SISD, SIMD, MIMD, MISD. La classe des machines MISD (Multiple Instruction Single Data) est sans intérêt pratique et celle des SISD (Single Instruction Single Data) correspond aux machines séquentielles. Je m'intéresse donc dans ce mémoire aux deux catégories SIMD et MIMD.

- SIMD (Single Instruction Multiple Data)

  C'est le type d'architectures massivement parallèle dont la CM-2 est l'un des exemples. Elle caractérise aussi l'architecture vectorielle comme le Cray 1. Dans ce modèle, plusieurs unités de traitement sont supervisées par une même unité de contrôle. Toutes les unités exécutent la même instruction (ou le même programme), mais opèrent sur des données distinctes.

- MIMD (Multiple Instruction Multiple Data)

  C'est l'architecture multiprocesseur. Dans ce cas, plusieurs processeurs possédant chacun leur propre unité de contrôle exécutent des programmes différents, par exemple: l'Encore-Multimax, la Sequent Balance, l'iPSC2, l'iPSC i860, le Cray X-MP, l'Alliant FX/8, l'IBM 3090 et la CM-5.

La puissance des machines parallèles est liée aux logiciels parallèles. La conception des programmes parallèles est la clé indispensable pour obtenir de bonnes performances de ces

architectures. Dans la section suivante, je présente une approche pour la programmation des machines parallèles.

## 1.2 Programmation parallèle

Pour obtenir des programmes parallèles, de nombreux programmeurs écrivent directement leurs programmes en utilisant un langage parallèle. Dans ce cas, le parallélisme est explicite et signalé à l'aide de primitives dans le programme. De très nombreux langages parallèles ainsi que des extensions de langages existants ont été proposé. Je présente brièvement trois exemples de langage parallèle qui ont des modèles de programmation différents et qui sont dédiés à trois types différents d'architectures parallèles: le langage OCCAM pour les architectures à base de Transputers, FORTRAN 90 pour les architectures vectorielles (ex. CRAY 1) et HPF, un FORTRAN étendu, pour multiprocesseur SIMD et MIMD.

### 1.2.1 OCCAM

Le langage Occam a été conçu à partir de CSP (*Communicating of Sequential Process*) pour la programmation parallèle des architectures basées sur les *transputers* [Kerr87]. Un transputer contient un processeur, une mémoire et des liens de communication. Il intègre un ordonnanceur gérant des processus par micro-code. Les communications entre transputers sont réalisées par l'intermédiaire des liens qui gèrent les envois et réceptions des données sur le réseau de transputers.

L'unité parallèle de ce langage est le processus (séquence d'opérations). Plusieurs processus peuvent être exécutés en parallèle. Les processus communiquent par envoi et réception de messages sur des canaux. Ces échanges sont bloquants et assurent donc une synchronisation en supplément d'un transfert de valeur. Les primitives principales du parallélisme sont les suivantes:

- **PAR**: exécution parallèle
- **SEQ**: exécution séquentielle
- **c? var**: réception d'une variable **var** sur un canal **c**
- **c! expr**: envoi de la valeur d'une expression **expr** sur un canal **c**

Le parallélisme de contrôle est donc le modèle sous-jacent. Ce modèle est adapté aux multiprocesseurs MIMD mais il ne permet pas d'exprimer naturellement des calculs scientifiques. De plus, OCCAM est une version de CSP qui oblige le programmeur à savoir si les processus qu'il manipule vont se trouver sur le même processeur physique ou non. Il est donc, en général, inapproprié pour coordonner un grand nombre de processus.

Voici un exemple de deux processus communiquant:

```
PROC p1(CHAN com)
      VAR x:
      SEQ
      …
      x:= …
      com ! x:
      …
PROC p2(CHAN com)
      VAR y:
      SEQ
```

```
        …
        com ? y
        …:= y
        …
CHAN comm :
PAR
        p1(comm)
        p2(comm)
```

Les processus p1 et p2 sont exécutés en parallèle. P1 calcule la valeur d'une variable x et envoie cette valeur au canal *comm*, p2 reçoit cette valeur par le même canal et l'utilise dans le processus.

### 1.2.2 FORTRAN

Le FORTRAN 90 est un langage conçu pour les machines vectorielles. Il est compatible avec le FORTRAN 77, mais possède des opérations et des fonctions plus riches. FORTRAN 90 permet le *traitement des tableaux en parallèle* [MeRe89].

Les opérations vectorielles principales sont:

- L'affectation de section de tableau

Par exemple:

```
REAL DIMMENSION (:;:):: A, B, C
...
A = B + C
B(1:N,1) = A(1, 1:N)
```

- Les opérations logiques sur les tableaux pour les instructions *IF* et *WHERE*

Par exemple:

```
WHERE (A > 0.0)
A = B * C
END WHERE
```

- Les fonctions intrinsèques sur les tableaux telles que: la valeur maximale ou minimale des éléments d'un tableau *MAXVAL(ARRAY)* et *MINVAL(ARRAY)*, le produit des éléments de deux vecteurs *DOTPRODUCT(ARRAY1, ARRAY2)*, la somme des éléments de deux vecteurs *SUM(ARRAY1, ARRAY2)*, etc.

Le parallélisme semi-explicite est contenu dans l'expression et les opérations vectorielles du langage.

### 1.2.3 HPF

HPF (High Performance Fortran) est un langage de programmation data-parallèle [HPF97]. HPF est une extension de Fortran 90 pour la programmation à parallélisme de données. Il est basé sur la distribution explicite des données, et doit permettre l'obtention de bonnes performances sur les machines MIMD et SIMD à mémoire non uniforme tout en préservant la portabilité. HPF a étendu Fortran sous plusieurs aspects: distribution de données, instruction parallèle, intrinsèques et librairies, etc.

Afin de permettre d'expliciter les calculs parallèles, HPF offre une nouvelle instruction FORALL et une nouvelle directive INDEPENDENT. L'instruction FORALL est une extension de l'affectation à un tableau du Fortran 90. La construction FORALL regroupe plusieurs affectations qui seront exécutées en parallèle les unes après les autres sur tout le domaine d'itération. La directive INDEPENDENT informe le compilateur que les opérations qui suivent peuvent être exécutées en parallèle. Elle peut précéder une boucle DO ou FORALL.

Voici un exemple de construction FORALL et un exemple de directive INDEPENDENT.

- Exemple 1:

```
do i = 1,3                          forall i = 1,3
ga(i) = da(i)                       ga(i) = da(i)
gb(i) = db(i)                       gb(i) = db(i)
enddo                               endforall
```

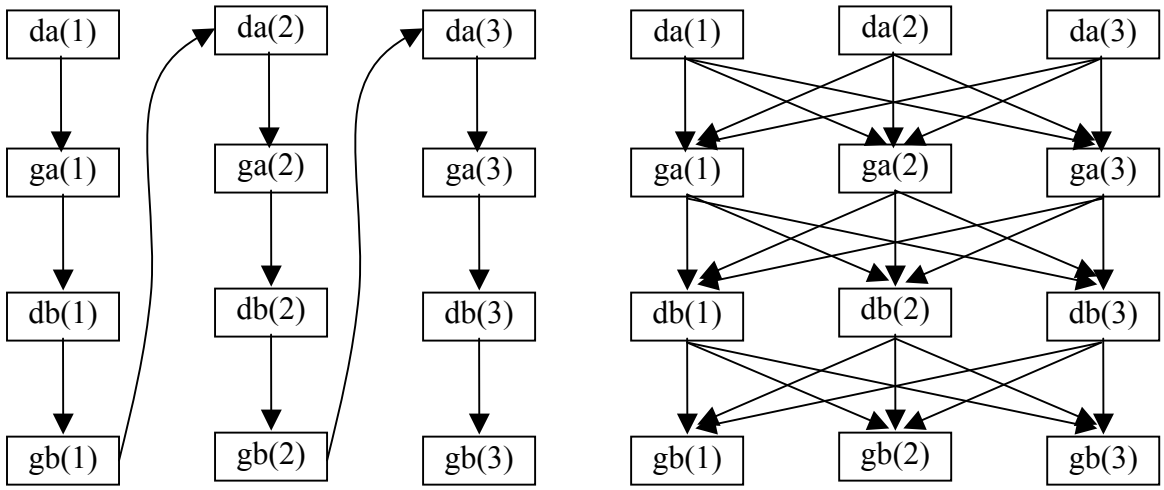


*Figure 1 : Comparaison de la sémantique d'une exécution d'une boucle et d'un FORALL*

Dans le cas d'une boucle, pour la première itération, la partie droite de la première affectation (da(1)) est calculée puis affectée à la partie gauche (ga(1)), puis la partie droite de la deuxième affectation (db(1)) est calculée puis affectée à la partie gauche (gb(1)), et ainsi de suite pour toutes les itérations. Dans le cas d'un FORALL, le graphe illustrant l'exécution est plus compliqué : les parties droites de toutes les itérations de la première affectation (da(1), da(2), da(3)) sont calculées, puis elles sont affectées aux parties gauches de toutes les itérations de la première affectation, etc. La sémantique d'exécution du FORALL permet d'implanter le calcul des parties droites de la première itération en parallèle, puis toutes les affectations de la première instruction, puis de la même façon pour la deuxième instruction, avec la nécessité d'effectuer une synchronisation entre toutes ces phases de calcul. Le FORALL de la figure 1 est équivalent aux deux affectations vectorielles de Fortran90 :

```
ga = da
gb = db
```

- Exemple 2:

```
!HPF$ INDEPENDENT
DO i=1,100
   A(P(i)) = B(i)          ! P est une permutation
ENDDO
```

La directive INDEPENDENT affirme que toutes les itérations de la boucle i sont indépendantes.

La programmation parallèle impose aux programmeurs la responsabilité de l'identification du parallélisme et sa spécification en langage parallèle. C'est une tâche assez complexe même si le langage possède toutes les constructions parallèles nécessaires.

De nombreux programmes écrits en FORTRAN 66 ou en FORTRAN 77 sont encore utilisés dans le monde du calcul scientifique. Leur traduction manuelle en codes parallèles n'est pas envisageable, même pour de petites applications, sachant que toutes les relations de dépendance doivent être calculées et vérifiées. La section suivante présentera la notion de dépendance de données qui est très importante pour la transformation afin d'obtenir le parallélisme maximum.

## 1.3 Dépendance de données

La sémantique d'un programme écrit en langage impératif, comme Fortran ou C, définit un ordre d'exécution séquentiel des instructions du programme. Cependant, généralement, plusieurs ordres différents d'exécution des instructions peuvent aboutir à l'obtention d'un même résultat. Par exemple, l'ordre des deux instructions A = B * C et D = E + F peut être échangé sans affecter le résultat du programme. Deux instructions peuvent être exécutées en parallèle si ces deux instructions n'accèdent pas au même emplacement mémoire. Dans ce cas, on dit que les instructions sont indépendantes, sinon on dit qu'il existe une relation de dépendance entre ces deux instructions. Toutes les relations de dépendance existant dans un programme spécifient les contraintes sur l'ordre d'exécution des instructions qui permettent de conserver la sémantique du programme.

Dans cette section, je présente le concept de dépendance et en donne une définition, tout particulièrement pour le cas du nid de boucles car elles contiennent l'essentiel du parallélisme implicite.

### 1.3.1 Concept de dépendance de données

Je décris, dans cette section, les trois types de dépendance définis à l'origine par Kuck [Kuck78] et le graphe de dépendance de données.

*Définition d'une dépendance:*

On dit que l'instruction T dépend de l'instruction S (noté S $\delta$ T) si il existe une instance t de T, une instance s de S et un emplacement mémoire M tels que:

- s et t référencent M et au moins une instance le modifie
- selon l'ordre d'exécution séquentiel, s s'exécute avant t

On distingue trois types de dépendance, conditionnant l'exécution parallèle d'un programme.

La dépendance de S vers T est:

- Une *dépendance de flot* si s modifie M et t lit M. Elle est notée S $\delta^f$ T
- Une *anti-dépendance* si s lit M et t modifie M. Elle est notée S $\delta^a$ T
- Une *dépendance de sortie* si s et t modifient M. Elle est notée S $\delta^o$ T

Un quatrième type de dépendance, appelé la *dépendance de l'entrée*, correspondant à la lecture de M par s et t est utilisé pour l'optimisation de l'allocation des données en mémoire.

Exemple:

Soit le programme écrit en FORTRAN :

```
S1 :        A = 0
S2 :        B = A
S3 :        C = A + D
S4 :        D = 2
S5 :        D = 0
```

Il y a les dépendances suivantes:

$S_1\ \delta^f\ S_2$ , $S_1\ \delta^f\ S_3$ , $S_3\ \delta^a\ S_4$ , $S_3\ \delta^a\ S_5$ , et $S_4\ \delta^o\ S_5$ .

Ces dépendances sont aussi l'ordre d'exécution qu'il va falloir respecter dans les transformations. Cet ordre peut être représenté par un graphe où les arcs représentent les dépendances et où il y a un sommet par opération. Nous appelons ce graphe le graphe des dépendances de données (DDG).

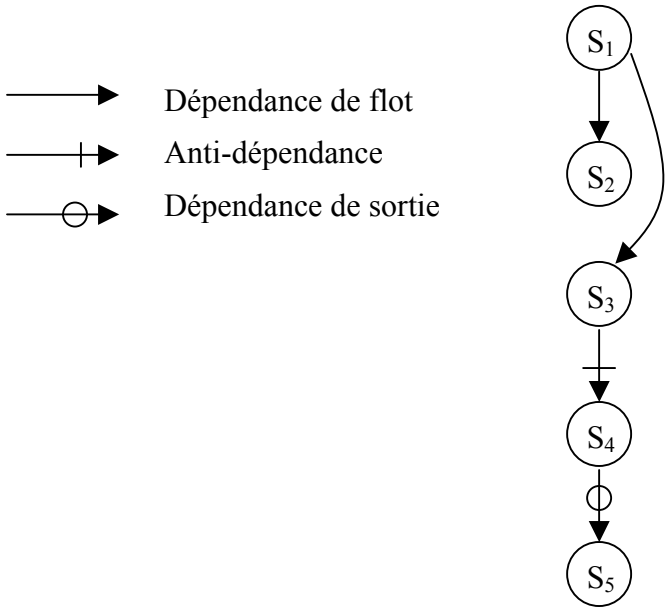


*Figure 2: Graphe de dépendance*

### *1.3.2 Représentation des dépendances de données*

Nous allons nous intéresser dans cette partie aux différentes approximations des dépendances utilisées pour étiqueter les arcs d'un DDG.

Soit le programme P :

```
Do I = 1, n
      Do J= 1, n
S           T(I,J) = T(3*I, J+1)
      Enddo
Enddo
```

***Itérations de dépendances:***

L'abstraction DI(L) est exactement l'ensemble d'itérations de dépendances entre les instances d'instruction de boucles.

$$DI(L) = \left\{ (\vec{i}, \vec{j}) \,\middle|\, \exists S_1, S_2 \in L, S_1(\vec{i}) \,\delta\, S_2(\vec{j}) \right\}$$

Pour le programme P, DI est représenté dans la figure 3. La version paramétrique d'itérations de dépendances peut être décrite comme suivante:

$$DI(P) = \left\{ ((i,j),(i',j')) \,\middle|\, i' = 3i,\ j' = j+1, 1 \le i \le n, 1 \le j \le n \right\}$$

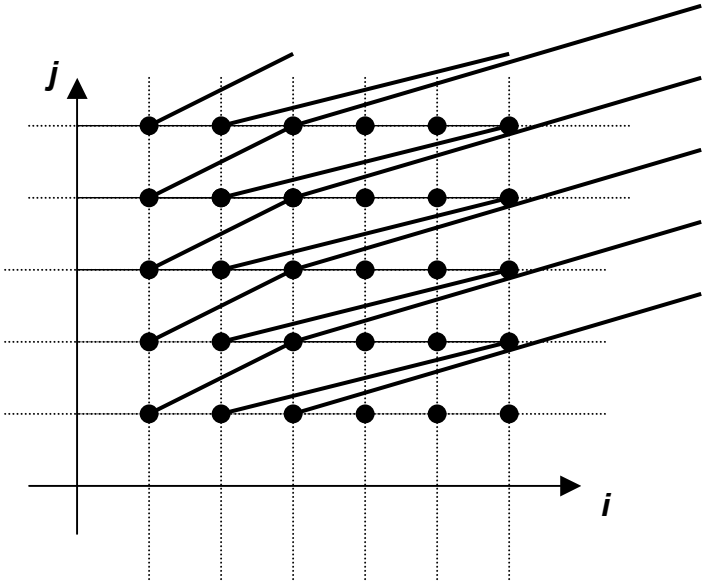


*Figure 3: Itérations de dépendances*

Au lieu de représenter exactement toutes les paires $(\vec{p},\vec{q}) \in P^2$ induisant une dépendance, on exprime l'ensemble D des valeurs $\vec{d} = \vec{p} - \vec{q} \in P$ (sur la partie commune des vecteurs) que l'on appellera l'ensemble des vecteurs de distance ou encore l'ensemble des vecteurs de dépendance. L'utilisation d'un vecteur de distance $\vec{d}$ permet de représenter toutes les dépendances entre itérations $\vec{p}$ et $(\vec{p} + \vec{d})$. Elle reflète explicitement le pas de dépendance entre les itérations mais ne tient pas compte de l'origine des dépendances.

***Vecteurs de distance:***

$$D(L) = \left\{ \vec{d} \,\middle|\, \exists (\vec{i}, \vec{j}) \in DI(L), \vec{d} = \vec{j} - \vec{i} \right\}$$

Pour le programme P, le vecteur de distance est décrit comme suit et représenté dans la figure 4

$$\begin{cases} d_i = 2i, 1 \le i \le n, 1 \le \mathrm{i} + \mathrm{d_i} \le n \\ d_j = 1, 1 \le j \le n, 1 \le j + \mathrm{d_j} \le n \end{cases}$$

$$D(P) = \left\{ (2k,1) \,\middle|\, 1 \le k \le n \right\}$$

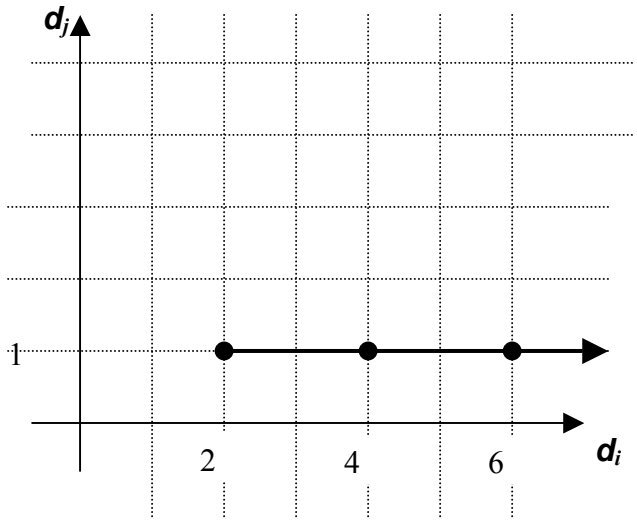


*Figure 4: Vecteurs de distance de dépendances*

En comparaison avec la représentation de DI dans la figure 2, la différence espace (i,j) est remplacé par ($d_i$, $d_j$), comparé avec DI, l'utilisation de D réduit l'espace mémoire nécessaire pour stocker les dépendances uniformes (dépendance constante). Par exemple D = {2} représente l'ensemble infini d'itérations de dépendances constantes dont la version paramétrique est:

$$DI = \left\{ ((\vec{i}, \vec{j}) \mid \vec{\mathrm{j}} = \vec{i} + 2, \vec{\mathrm{i}} \geq 0 \right\}$$

Dans plusieurs cas, le vecteur de distance de dépendances n'est pas constant, et pour cette raison, des approximations comme le polyèdre de dépendance, le cône de dépendance, le vecteur de direction de dépendance et le niveau de dépendance sont introduites.

***Polyèdre de dépendance:***

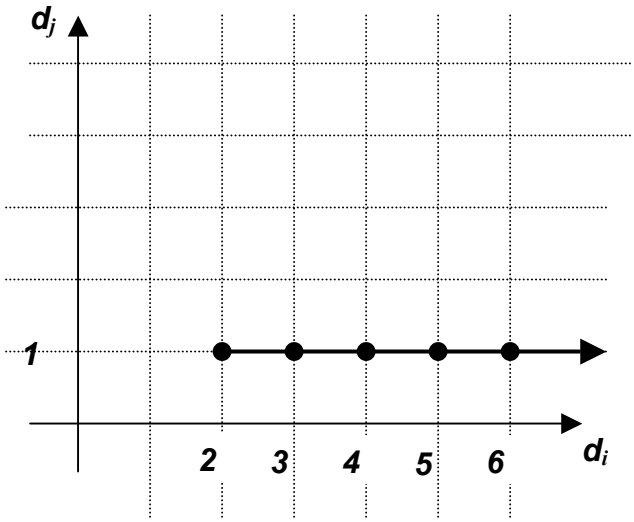


*Figure 5: Polyèdre de dépendance*

Le polyèdre de dépendance DP(L) calcule les approximations entre l'ensemble des vecteurs de distances de dépendances et l'ensemble des points qui sont combinaisons convexes des vecteurs D(L). Il calcule l'ensemble des points entiers de l'enveloppe convexe (convex hull) de D(L).

$$DP(L) = \left\{ \vec{\mathrm{v}} = \sum_{1}^{k} \lambda_i \vec{d}_i \in Z^{\mathrm{n}} \;\middle|\; \vec{d}_i \in \mathrm{D(L)}, \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i = 1 \right\}$$

Bien que DP(L) soit une approximation de D(L), il contient toutes les informations utiles pour bien appliquer aux transformations de re-ordonnancement comme abstraction D. Figure 4 illustre DP(L) pour le programme P. La l'enveloppe contient de nouveaux vecteurs (3,1), (5,1), etc.…

***Cône de dépendance:***

Le cône de dépendance DC(L) approxime D(L) avec l'ensemble des points qui sont combinaisons linaires positives de vecteurs D(L). Il est défini comme suit:

$$DC(L) = \left\{ \vec{\mathrm{v}} = \sum_{1}^{k} \lambda_i \vec{d}_i \in Z^{\mathrm{n}} \;\middle|\; \vec{d}_i \in \mathrm{D(L)}, \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i \geq 1 \right\}$$

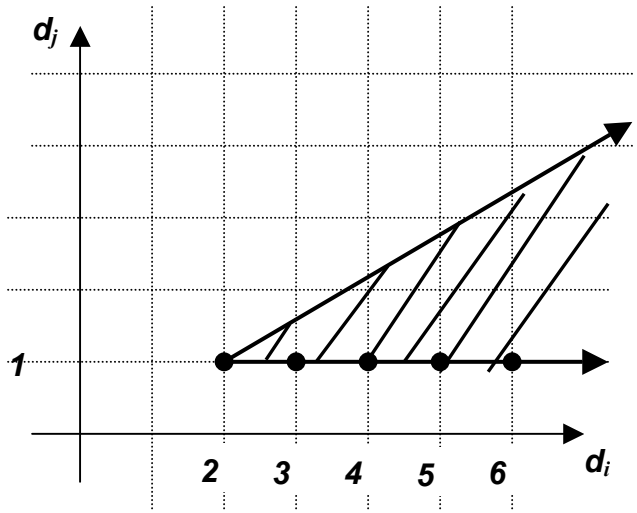


*Figure 6: Cône de dépendance*

L'avantage principal des deux abstractions DP et DC est que, dans plusieurs cas, une seule structure est nécessaire pour la représentation des informations contenues dans D(L). Le gain d'espace mémoire est notable particulièrement quand l'ensemble de dépendance est fini ou est grand avec des dépendances non-uniformes. DP et DC sont plus faciles à utiliser que D(L) pour le calcul des partitions de boucles.

***Vecteur de direction de dépendance:***

Les abstractions DP(L) et DC(L) sont une approximation de D(L). Ces deux abstractions contiennent les informations utiles pour les transformations qui réordonnent les ensembles d'itérations. Cependant, les transformations comme l'échange de boucles qui modifient

seulement le signe du vecteur de distance de dépendance, ne nécessitent pas les informations de distance exactes. Des abstractions comme DDV(L) et DL(L) concernant donc seulement le signe ou le niveau du vecteur de dépendance ont été introduites.

Chaque élément du vecteur de direction de dépendance est choisi parmi {<, =, >}. Les autres éléments comme ≤, *, ≥ peuvent être utilisés pour récapituler deux ou trois éléments du DDV.

Il est défini comme suit:

$$DDV(L) = \left\{ (\psi_1, \psi_2, ..., \psi_n) \middle| \begin{array}{l} \exists S_1, S_2 \in L, S_1(\vec{p})\, \delta^*\, S_2(\vec{q}), \mathrm{p_k}\ \psi_\mathrm{k}\ \mathrm{q_k}, \\ (1 \le k \le n), \psi_i \in \{<, =, >\} \end{array} \right\}$$

Pour le programme P, DDV(P) = {(<, = >)}, il est représenté dans la figure 7.

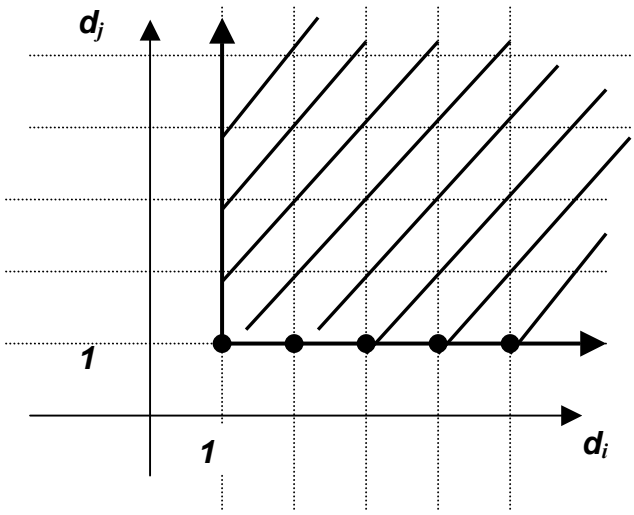


*Figure 7: Vecteur de direction de dépendance*

***Niveau de dépendance:***

Le niveau de dépendance a été introduit par Allen et Kennedy [AlKe87] pour la vectorisation et la parallélisation de programme. Il donne le niveau de puits de la boucle la plus extérieure *l* portant la dépendance, quand le composant du vecteur de dépendance est positif. Pour conserver la sémantique du programme, la boucle *l* doit être gardée séquentielle. Ainsi, l'ordre lexicographique est conservé et toutes les boucles intérieures peuvent être parallélisées si aucune autre dépendance n'existe. DL(L) est défini comme suit:

$$DL(L) = \left\{ \mathrm{k} \middle| \exists (\psi_1, \psi_2, ..., \psi_n) \in \mathrm{DDV}(L), 1 \le i \le k-1, \wedge \psi_i\ \mathrm{is} = \wedge \psi_k is < \right\}$$

Pour le programme P, DL(P) = {1}, le niveau de dépendances est représenté dans la figure 8.

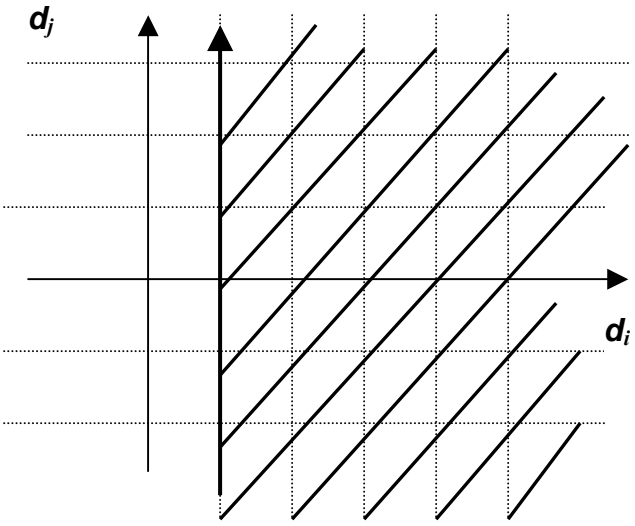


Figure 8: Niveau de dépendance

## 1.4 Dépendance de données dans PIPS

### 1.4.1 PIPS

PIPS est un Paralléliseur Interprocédural de Programmes Scientifiques source-à-source de FORTRAN qui a été développé au Centre de Recherche en Informatique de l'Ecole des Mines à Fontainebleau depuis 1988 [IJT91]. PIPS transforme des programmes écrits en FORTRAN 77 en programme FORTRAN 90 ou FORTRAN parallèle. Les boucles DO séquentielles parallélisables sont remplacées par des instructions vectorielles FORTRAN 90 ou par des constructions de type HPF. Les trois caractéristiques principales de PIPS sont:

- L'interprocéduralité qui est prise en compte dans les phases d'analyse du programme.
- Une analyse sémantique sophistiquée, basée sur les pré-conditions et les régions, qui permet l'obtention de résultats très précis lors du calcul des effets, des dépendances, de l'estimation de la complexité statique du programme ou encore pour la sélection d'une transformation.
- Une efficacité suffisante pour effectuer des expériences sur des programmes réels.

### 1.4.2 Calcul des use-def chains

La phase *Chains* de PIPS calcule les *use-def chains*, caractérisant les variables scalaires et les références aux éléments des tableaux lus ou modifiés par le programme, et produit le graphe des *chains* qui correspond à une première ébauche du graphe de dépendance. Dans ce graphe initial, un arc entre deux références à un même tableau spécifie qu'il existe une dépendance sur la variable tableau, les indices du tableau référencés n'étant pas pris en compte (des arcs entre deux éléments T(2) et T(3) peuvent exister). Une phase d'analyse des dépendances est ensuite effectuée afin d'éliminer autant d'arcs que possible. Pour les variables scalaires, tous les arcs sont conservés. Pour les variables tableau, un test de dépendance est appliqué. Les arcs correspondant à de *fausses* dépendances sont éliminés.

### 1.4.3 Représentation du graphe de dépendance

Après avoir effectué la phase d'analyse des dépendances afin d'éliminer des arcs dans les *use-def chains*, PIPS obtient le graphe de dépendances de données. Il est affiché comme suit:

Soit le programme:

```
…
6     real x(10,10,10)
7     real s(10)
8     do i = 1, 10
9           s(i) = 0
10          do j = 1, 10
11                do k = 1, 10
12                      s(i) = s(i) + x(i,j,k)
13                enddo
14          enddo
15    enddo
...
```

Affichage du graphe de dépendances de PIPS:

```
12 12 R W <S(I)> - <S(I)> levels(2,3) ddv(=,<=,*)
12 12 W R <S(I)> - <S(I)> levels(2,3) ddv(=,<=,*)
12 12 W W <S(I)> - <S(I)> levels(2,3) ddv(=,<=,*)
12 11 R W <K> - <K> levels(1,2)
12 10 R W <J> - <J> levels(1)
11 11 W W <K> - <K> levels(1,2)
10 10 W W <J> - <J> levels(1)
9 12 W R <S(I)> - <S(I)> levels(2)
9 12 W W <S(I)> - <S(I)> levels(2)
```

Explication de la première ligne de l'affichage:

```
12 12 R W <S(I)> - <S(I)> levels(2,3) ddv(=,<=,*)
```

*12 12* : ce sont deux indices de deux instructions qui sont dépendants, dans ce cas ils sont pareils

R W : nous dit que c'est une anti-dépendance

*<S(I)> - <S(I)>* : ce sont deux variables tableaux qui sont dépendants

*levels(2,3)* : niveaux de dépendance

*ddv(=,<=,*)* : vecteurs de direction de dépendance

Cet affichage du graphe de dépendance est représenté sous une forme texte qui n'est pas très lisible. Le chapitre suivant présente la visualisation du graphe de dépendance que j'ai implémentée.

## 1.5 daVinci, outil d'affichage des graphiques

Afin de représenter graphiquement le graphe de dépendance ou le graphe de flot de contrôle sur X-Window, j'utilise *daVinci*, un outil d'affichage des graphiques.

*daVinci* est un outil X-Window développé par l'Université de Bremel pour visualiser automatiquement les graphes directionnels. Le format du fichier *daVinci* appelé *la représentation de terme* est composé de caractères ASCII.

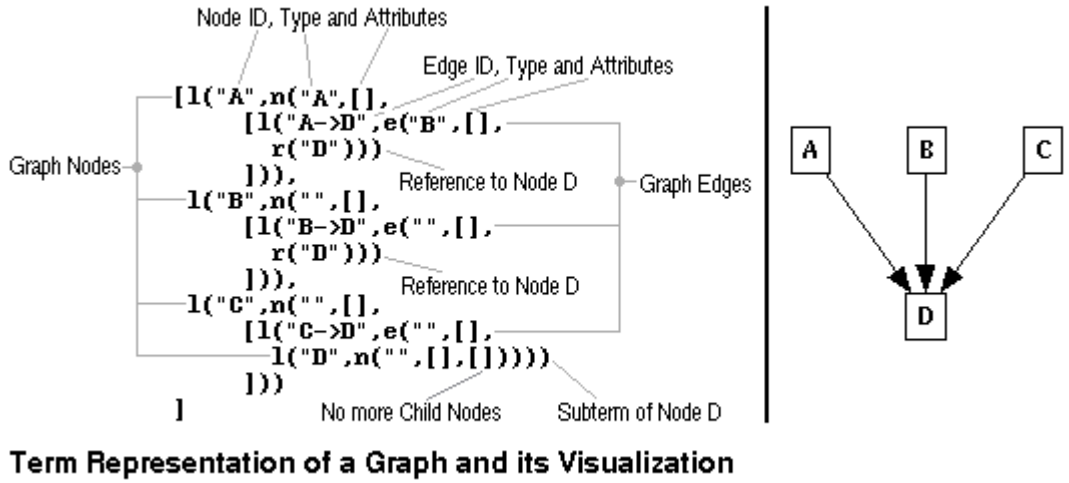


*Figure 9: Terme de représentation et un exemple de graphe affiché par daVinci*

La figure 10 démontre un exemple de fichier *daVinci* de représentation d'un graphe et son affichage dans *daVinci*

```
[
 l("Node A",n("Module",[a("COLOR","#f31d8b"),
                        a("OBJECT","Node A"),
                        a("_GO","ellipse"),
                        a("FONTFAMILY","helvetica")],
    [
    l("Edge A->B",e("",[a("EDGECOLOR","red")],
     l("Node B",n("anything",[a("COLOR","light steel blue"),
                              a("BORDER","double"),
                              a("OBJECT","Node B\nThis node has two
lines")],
        [
        l("Edge B->D",e("",[a("EDGEPATTERN","dotted"),
                            a("_DIR","inverse")],
         r("Node D")))
        ])))),
    l("Edge A->C",e("",[a("EDGEPATTERN","dotted"),
                        a("EDGECOLOR","#0054aa")],
     l("Node C",n("anything",[a("_GO","icon"),
                              a("ICONFILE","bkb.xbm"),
                              a("COLOR","gold"),
                              a("OBJECT","Node C"),
                              a("FONTSTYLE","italic")],
        [
        l("Edge C->D",e("anything",[a("EDGECOLOR","#00aaaa"),
                                    a("_DIR","both")],
         l("Node D",n("Module",[a("OBJECT","Node D"),
                                a("_GO","text"),
                                a("FONTFAMILY","times"),
                                a("FONTSTYLE","italic")],[])))),
        l("Edge C->F",e("anything",[a("EDGEPATTERN","double")],
         r("Node F")))
        ])))),
    l("Edge A->E",e("",[a("EDGEPATTERN","dashed"),
                        a("_DIR","none")],
     l("Node E",n("anything",[a("COLOR","LightGray"),
                              a("FONTFAMILY","courier"),
                              a("FONTSTYLE","normal"),
                              a("BORDER","double"),
                              a("OBJECT","Node E"),
```

```
                                a("_GO","rhombus")],
        [
        l("Edge E->F",e("anything",[a("EDGEPATTERN","thick")],
         l("Node F",n("anything",[a("COLOR","#00dddd"),
                                  a("OBJECT","Node F"),
                                  a("HIDDEN","true")],
            [
            l("Edge F->G",e("anything",[],
             l("Node G",n("anything",[a("OBJECT","Node G")],[])))))
            ])))))
        ])))
    ]))
]
```

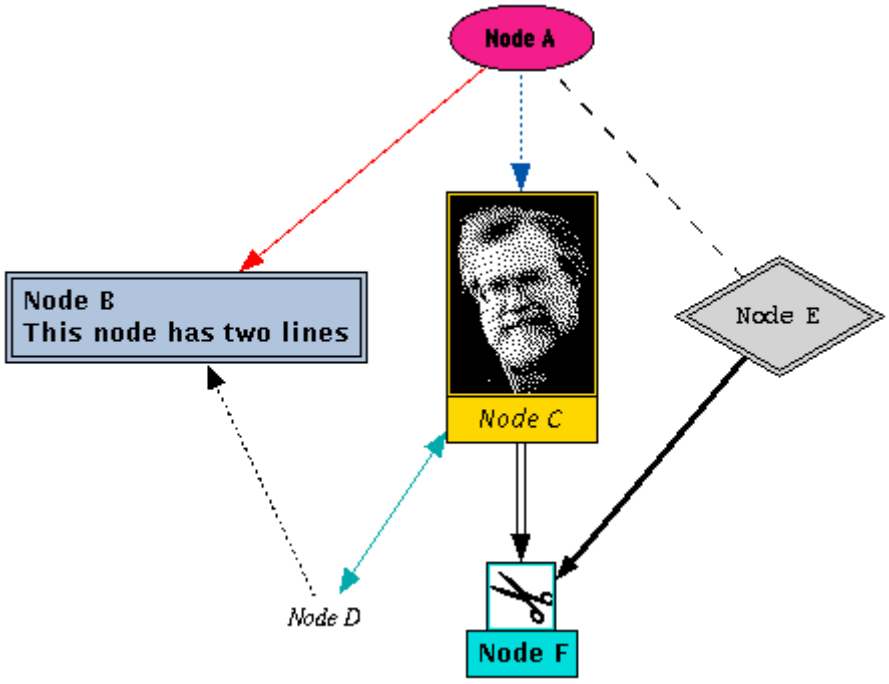


*Figure 10 : Exemple de représentation d'un graphe et son affichage pour daVinci*

# Chapitre II. Graphe de dépendance

Nous avons déjà vu la représentation du graphe de dépendance de PIPS dans le chapitre précédent. Une phase manquante dans PIPS est le filtrage du graphe de dépendance sur une ou plusieurs variables et sa représentation graphique. Dans ce chapitre, je présente l'implémentation de cette phase.

## *4.1 Filtrage du graphe de dépendance*

Le graphe de dépendance d'un programme est difficile à voir lorsque le programme est grand, un filtrage sur certaines variables et la visualisation du graphe de dépendance est très nécessaire.

Afin de filtrer le graphe de dépendance sur une ou plusieurs variables, nous devons parcourir toutes les dépendances dans le graphe et voir si cette dépendance appartient à la liste de variables à filtrer ou pas. L'algorithme est décrit ci-dessous.

*Algorithme:*

```
Fonction print_filtered_dg_or_dvdg
        Obtenir le graphe de dépendance G
        Obtenir la liste des variables à filtrer list_of_variables
        Construire un nouveau graphe vide GF (graphe filtré)
        Pour chaque vertex Ve du graphe G Faire
                Pour chaque successeur Su associé au vertex Ve Faire
                        Obtenir la variable V référencée par le successeur Su
                        Si V est dans liste list_of_variables Faire
                                Construire le graphe GF en ajoutant le successeur Su
                        FinSi
                FinPour
        FinPour
        Afficher le graphe GF sous forme texte ou sous forme graphique
Fin
```

## *4.2 Visualisation du graphe de dépendance*

Comme nous l'avons déjà présenté dans le chapitre précédent, nous utilisons l'outil d'affichage *daVinci* pour représenter le graphe de dépendance sous forme graphique. Il y a problème qui est que *daVinci* ne supporte pas la décoration des arcs. Cette fonction est indispensable car nous devons présenter les propriétés d'une dépendance. On peut régler ce problème en introduisant un nouveau sommet qui a un type différent des autres. Cette solution est illustrée par la figure 11.

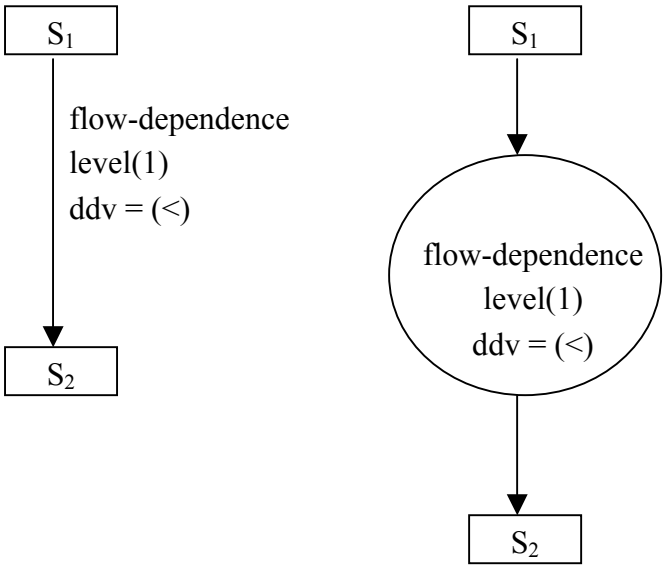


*Figure 11: Décoration des arcs en utilisant daVinci*

## 4.3 Expérimentations

Ce module a été implémenté et validé. La validation est effectuée avec plusieurs codes écrits par l'auteur et certains vrais codes du benchmark SPEC95.

Pour plus de détails sur la structure de données voir la partie *Annexe*.

Dans l'exemple ci-dessous, l'entrée est un programme Fortran, la sortie est l'affichage du graphe de dépendance sous forme texte et sous forme graphique, filtré sur la variable S et B.

**ENTREE**

```
      program trace
      COMMON /X/ S
      real a, b
            a = 0
            do i=1,10
                  b = 2 * a
                  a = a + 1
            enddo
            S = 1
            do i=1,10
                  call subA(S)
                  call subB(S)
            enddo
            if (S.GE.1) then
                  a = a * 2
            endif
      END
      subroutine subA(N)
      Integer N
C     COMMON /X/ S
            N = 1
            call subC
            return
      end
      subroutine subB(M)
      Integer*2 M
      real T
C     COMMON /X/ S
```

```
            T = M+1
            call subD
            return
      end
      subroutine subC
            return
      end
      subroutine subD
            return
      end
```

**SORTIE:** Graphe de dépendance filtré sur la variable S et B, affiché sous forme texte

```
12-<S>-R 11-<S>-W levels(1)
11-<S>-W 14-<S>-R
11-<S>-W 12-<S>-R levels(1,2)
11-<S>-W 11-<S>-W levels(1)
9-<S>-W 14-<S>-R
9-<S>-W 11-<S>-W
6-<B>-W 6-<B>-W levels(1)
```

**SORTIE:** Graphe de dépendance filtré sur la variable S et B, affiché sous forme graphique

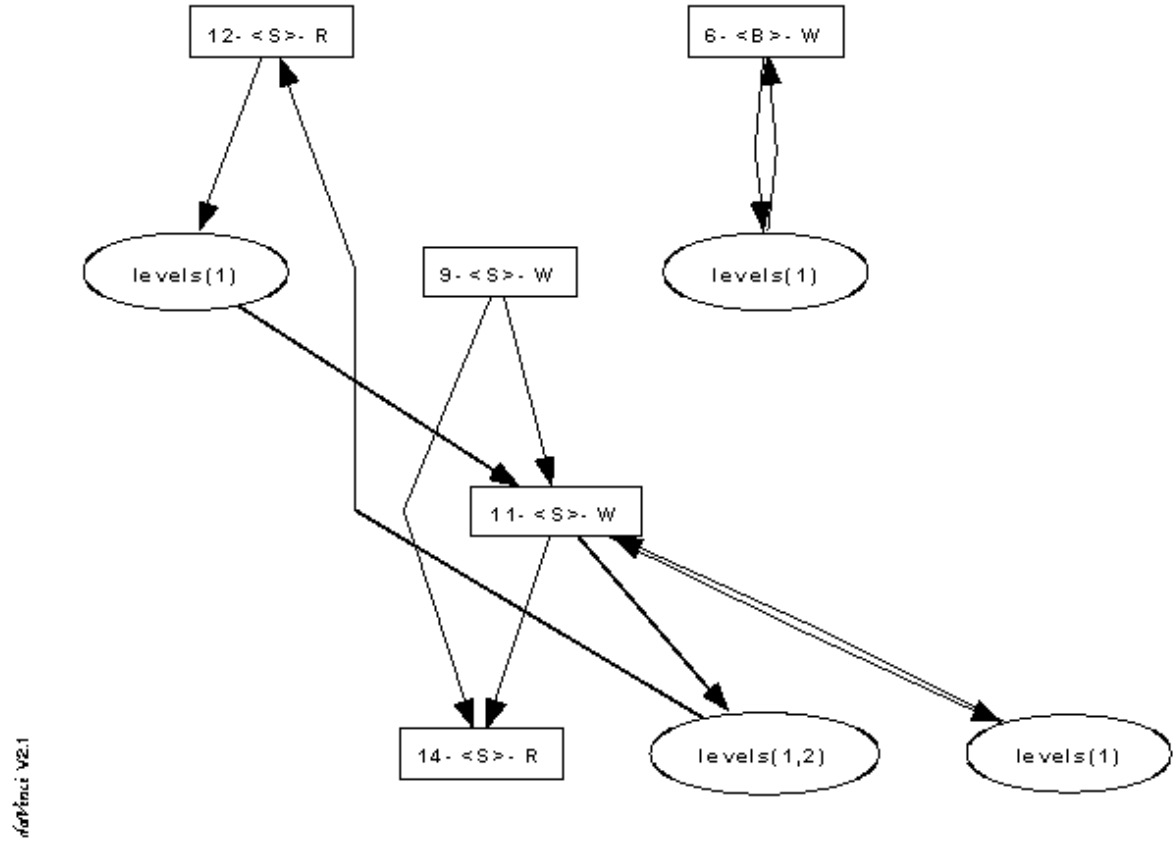


*Figure 12: Graphe de dépendance sous forme graphique*

# Chapitre III. Effets

Pour déterminer une dépendance on doit savoir si telle instruction écrit ou lit telle variable. Dans PIPS, le module *effets* effectue ce travail. PIPS a calculé les effets des instructions et aussi les effets cumulés d'une procédure. Une phase manquante dans PIPS est le filtrage des effets sur une ou plusieurs variables et sa représentation graphique. Dans ce chapitre, je présente l'implémentation de cette phase.

## *4.1 Filtrage des effets mémoire*

Comme pour le graphe de dépendance, lorsque le programme est grand, le nombre de variables est grand, et il devient très nécessaire de filtrer les effets mémoire sur certaines variables. La société EDF (Electricité de la France) nous demande aussi de visualiser les effets associés au graphe de flot de contrôle.

Afin de filtrer les effets mémoire, on doit parcourir toutes les instructions, examiner si cette instruction est un appel de procédure ou pas. Si oui, on doit obtenir ses effets cumulés, si non, on doit obtenir ses effets propres. On enfin doit construire le graphe de flot de contrôle en associant les effets. L'algorithme est expliqué ci-dessous:

*Algorithme:*

**Fonction** print_modul_icfg

```
Obtenir la liste des variables à filtrer list_of_variables
Construire un nouveau graphe vide GF (graphe filtré)
Pour chaque instruction S du module Faire
    Si S est un appel d'une procédure Faire
        Obtenir les effets cumulés de S
        Filtrer les effets obtenus sur list_of_variables
        DrapeauS = FALSE
        Pour chaque effet E dans les effets filtrés Faire
            Pour chaque argument A de procédure S Faire
                Si A est égal à la variable référencée par E Faire
                    Associer E à S
                    DrapeauS = TRUE
                    Sauter de Pour
                FinSi
            FinPour
        FinPour
        Si DrapeauS Faire Ajouter S au graphe GF
    Sinon
        Obtenir les effets propres de S
        Filtrer les effets obtenus sur list_of_variables
        Associer les effets filtrés à S
    FinSi
FinPour
```

```
        Afficher le graphe GF sous forme texte ou sous forme graphique
Fin
```

## 4.2 Visualisation des effets associés au graphe de flot de contrôle

Dans ce cas, dans le graphe à visualiser, il n'y a pas de décoration sur un arc, nous n'avons donc pas besoin d'introduire de nouveaux sommets.

## 4.3 Expérimentations

Cette phase a été implémentée et validée dans PIPS. La validation est effectuée avec plusieurs codes écrits par l'auteur et certains vrais codes du benchmark SPEC95.

Pour plus de détails sur la structure de données voir la partie *Annexe*.

Dans l'exemple ci-dessous, l'entrée est un programme Fortran, la sortie est l'affichage du graphe de flot de contrôle sous forme texte et sous forme graphique, avec les effets joints filtrés sur la variable MAIN:KMAX et MAIN:KT.

**ENTREE**

```
      program main
      COMMON /W/ KMAX,KT
      integer I
      call A(I)
      end

      subroutine A(I)
      COMMON /W/ KMAX,KT
      integer I
      KMAX = KMAX+2
      call B(I)
      call C3(I)
      call B(KMAX)
      return
      end

      subroutine B(I)
      COMMON /W/ KMAX,KT
      integer I
      KT = KT +3
      call C1(I)
      call C2(I)
      return
      end

      subroutine C1(I)
      COMMON /W/ KMAX,KT
      integer I
      call INC(KMAX)
      return
      end

      subroutine C2(I)
      COMMON /W/ KMAX,KT
      integer I
      call INC(KT)
      return
      end

      subroutine C3(I)
      COMMON /W/ KMAX, KT
```

```
      integer I
      call INC(KT)
      return
      end

      subroutine INC(K)
      integer K
      K=K+1
      return
      end
```

**SORTIE:** Graphe de flot de contrôle avec les effets filtrés sur la variable MAIN:KMAX et MAIN:KT

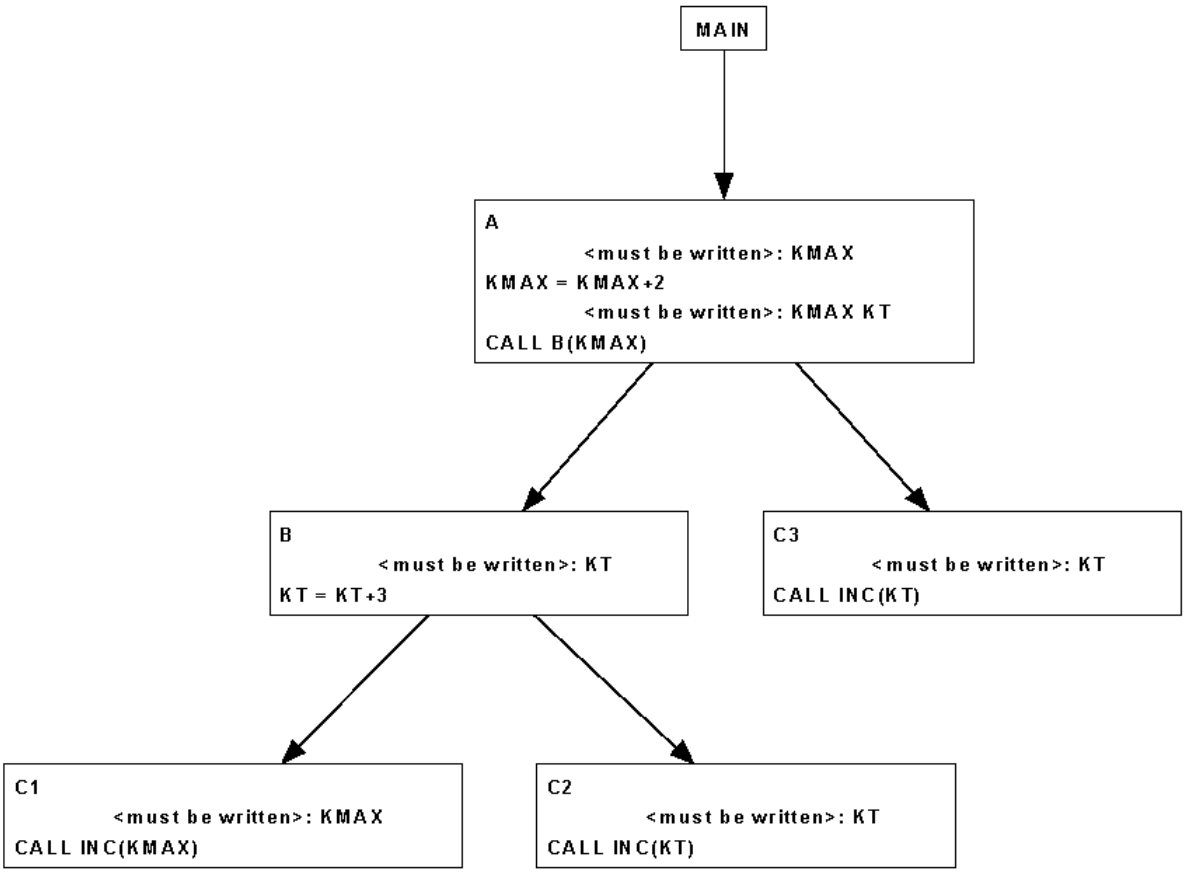


*Figure 13: Graphe de flot de contrôle avec des effets filtrés associés*

**SORTIE:** Graphe de flot de contrôle avec les effets filtrés sur la variable MAIN:KMAX

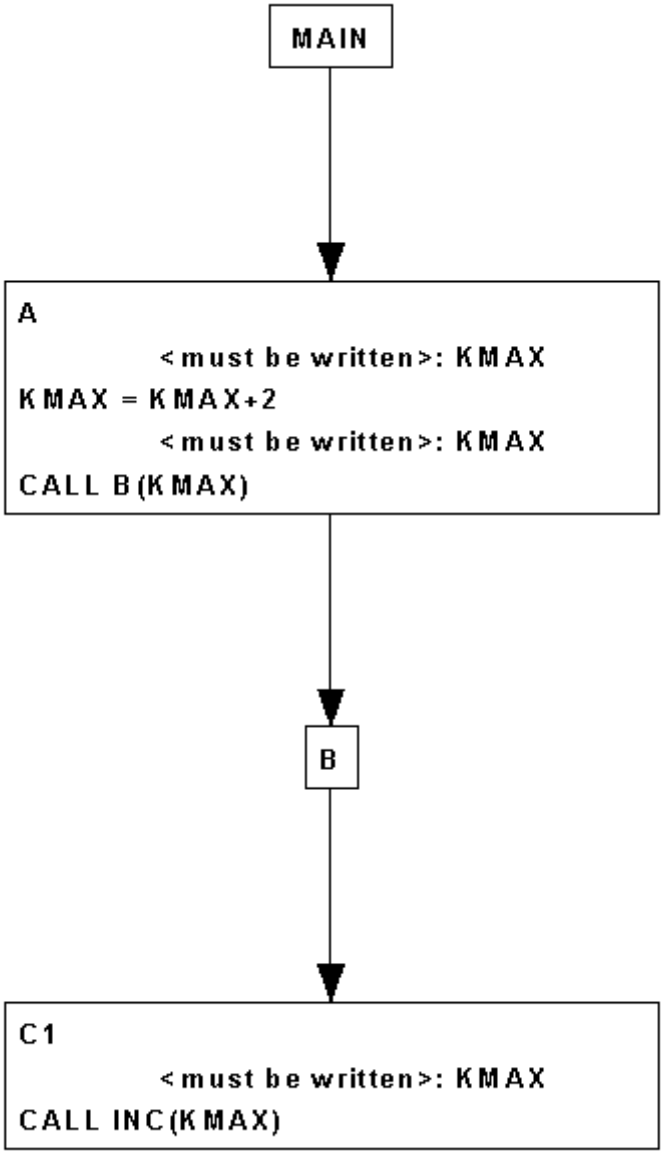


*Figure 14: Graphe de flot de contrôle avec des effets filtrés associés*

# Chapitre IV. Impact des alias

Normalement, le graphe de dépendance est construit pour une procédure. Mais au niveau inter-procédural, dans un programme, le graphe de dépendance d'une procédure change de temps en temps selon les appels de cette procédure ou bien la position de cette procédure dans le graphe de flot de contrôle. Ce chapitre consiste à étudier le changement du graphe de dépendance d'une procédure d'un appel à l'autre.

## *4.1 Problème d'alias*

Un alias apparaît lorsque deux ou plusieurs variables au même point du programme référencent un même emplacement mémoire. Il peut être détecté au moment de la compilation ou ne peut être détecté qu'au moment de l'exécution. Voyons les deux exemples ci-dessous.

Exemple 1:

```
PROGRAM ALIAS1                      SUBROUTINE SUB(X,Y)
INTEGER I, A(5), B                  INTEGER I, X(5), Y
DO I=1,5                            DO I = 1, 5
      A(I) = 2                            X(I) = Y * X(I)
ENDDO                               ENDDO
CALL SUB(A, B)                      END
CALL SUB(A, A(1))
WRITE (*,*), A
END
```

Dans cet exemple, le premier appel de procédure SUB (CALL SUB(A,B)) ne crée pas d'alias, le deuxième appel (CALL SUB(A, A(1)) crée un alias entre deux paramètres X et Y, cet alias peut être détecté au moment de la compilation. Les informations d'alias au moment de la compilation sont la clé pour analyser les dépendances de données et sont importantes pour l'analyse de programme, la parallélisation, la vérification, le déboguage et la compréhension de programme.

Exemple 2

```
PROGRAM ALIAS2                      SUBROUTINE SUB(V1, V2, N1, N2)
REAL A(100)                         REAL V1(N1), V2(N2)
READ *, I, J, M1, M2                DO I = 1, N1
CALL SUB(A(I), A(J), M1, M2)              V1(I) = 1
END                                 ENDDO
                                    DO I = 1, N2
                                          V2(I) = 2
                                    ENDDO
                                    END
```

Dans cet exemple, l'appel CALL SUB(A(I), A(J), M1, M2) peut créer un alias ou non, cela dépend de la valeur des variables I, J, M1, M2. Si [I, I+M1-1] et [J, I+M2-1] ne sont pas disjoints, il y aura un alias, si non il n'y a pas d'alias. Tout cela ne peut être détecté que lors de l'exécution.

Il y a plusieurs manières de créer des alias, cela varie selon les langages, la section suivante va présenter les possibilités de création des alias dans FORTRAN.

## 4.2 Possibilités de création des alias dans FORTRAN

Dans Fortran, tous les paramètres sont passés par référence, c'est-à-dire qu'une procédure appelée peut changer la valeur d'un argument en assignant une valeur à la variable formelle correspondante. Il y a trois manières de créer des alias dans FORTRAN.

- Passage des paramètres formels

Cette possibilité est déjà présentée dans les deux exemples précédents. Un argument actuel est passé à plusieurs paramètres formels lorsqu'on appelle une procédure.

- Alias entre un paramètre formel et une variable COMMON

Dans Fortran, une variable définie dans la section COMMON est une variable globale, on peut l'utiliser n'importe quand depuis n'importe où. Si on passe cette variable à une procédure qui a déjà utilisé cette variable, cela pourra créer un alias.

Exemple:

```
PROGRAM ALIASCOMMON
INTEGER X
COMMON /COM/ T
X = 0
CALL FOO(X)
CALL FOO(T)
END

SUBROUTINE FOO(X)
COMMON /COM/ T
T = T*X
END
```

Dans cet exemple, le premier appel ne crée pas d'alias, le deuxième appel crée un alias car la variable COMMON T est passée à la procédure FOO qui a déjà utilisé la variable COMMON T.

- Instruction EQUIVALENT

Dans Fortran, il y a une instruction EQUIVALENT qui force les différentes variables à être stockées dans le même emplacement mémoire.

Exemple

```
PROGRAM ALIASEQUIVALENCE
INTEGER X, Y
EQUIVALENCE (X, Y)
CALL FOO(X,Y)
END

SUBROUTINE FOO(X, Y)
X = Y
```

```
Y = Y + 1
END
```

Dans cet exemple, les deux variables X, Y sont stockées dans un même emplacement mémoire, elles sont aliasées.

## 4.3 Propagation inter-procédurale des alias

Afin de savoir si deux paramètres sont aliasés ou pas, on doit aller plus haut dans le chemin des appels pour savoir exactement d'ou viennent les paramètres. Toutes les informations concernant l'origine d'un paramètre sont déjà calculées dans PIPS, appelées propagation des alias. Elles sont stockées dans une structure alias_association.

structure alias_association

section: Emplacement mémoire pour stocker la variable

offset: position de la variable dans son emplacement mémoire

call_path: Le chemin des appels.

Example:

```
PROGRAM MAIN                          SUBROUTINE SUB2(V1, V2, L)
COMMON /COM/ W(50)                    COMMON /COM/ W(50)
REAL A(100), B(50)                    REAL V1(L), V2(L)
CALL SUB2 (W, B, 50)                  DO I = 1, L
CALL SUB1(A, 100)                           V1(I) = V2(I)
END                                   ENDDO
                                      END
SUBROUTINE SUB1(V,N)
REAL V(N)
READ *,M
IF (2*M.LE.N) THEN
      CALL SUB2(V, V(M+1), M)
ENDIF
END
```

premier appel:

section(V1) = COMMON:COM

section(V2) = ALIAS_SECTION_2

offset(V1) = 0

offset(V2) = 0

call_path(V1) = call_path(V2) = {MAIN:CALL SUB2(W,B,50)}

deuxième appel:

section(V1) = section(V2) = ALIAS_SECTION_1

offset(V1) = 0, offset(V2) = 4*L

call_path(V1) = call_path(V2) = {SUB1:CALL SUB2(V, V(M+1), M)}

Grâce à ces informations, on peut déterminer un alias entre deux variables au moment de la compilation ou au moment de l'exécution. Deux variables sont aliasées quand elles sont stockées dans un même emplacement mémoire, c'est à dire qu'elles ont la même *section* dans leur propagation et leurs positions (*offset* et longueur) dans cette section sont empilées l'une sur l'autre, et elles sont au même point du programme c'est à dire que leur *call_path* sont le même.

## 4.4 Impact des alias

Une transformation préserve la sémantique d'un programme si les dépendances de contrôle et de données sont respectées. Toutes les optimisations qui ne changent pas les dépendances du programme garantissent ne pas changer le résultat du programme.

L'influence de nouveaux arcs de dépendance créés par les alias dans le graphe de dépendance est importante parce qu'elle décide si une transformation de programme sur une procédure est possible. Si ces nouveaux arcs de dépendances sont redondants avec les arcs de dépendances existants, les alias n'ont pas d'impact sur les optimisations. Nous allons voir deux exemples ci-dessous:

```
      FOO(X,Y)
S1    X = X + 1
S2    T = Y
S3    Y = X
S4    X = T
      END
```

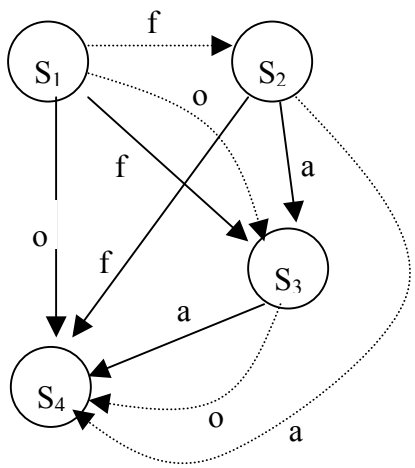


*Figure 15: Impact des alias*

Cette figure illustre le graphe de dépendance de données du programme quand il y a un alias entre deux paramètres X et Y. Les arcs pleins sont les anciens arcs, quand il n'y a pas d'alias entre X et Y, les arcs pointillés sont les nouveaux arcs, quand il y a un alias entre X et Y.

Voyons les nouveaux arcs: *output-dependence* de $S_1$ à $S_3$, *output-dependence* de $S_3$ à $S_4$, *anti-dependence* de $S_2$ à $S_4$, ils sont redondants avec les anciens arcs. Mais il y a un nouvel arc *flow-dependence* de $S_1$ à $S_2$, il n'est pas redondant avec les anciens arcs, cela crée une nouvelle contrainte d'ordonnancement. L'alias entre X et Y peut donc causer un vrai problème de transformation de programme.

```
      FOO(X,Y)
S1    T = X
S2    X = Y
S3    Y = T
      END
```

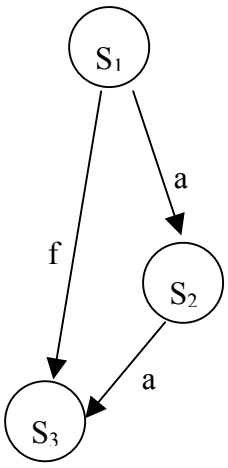


*Figure 16: Le graphe de dépendance totalement ordonné*

Dans ce graphe de dépendance, les sommets sont totalement ordonnés, c'est pourquoi aucun alias ne crée d'impact sur le graphe de dépendance.

La question qui se pose, c'est quand un alias influence la sémantique du programme. La section suivante va présenter les études des impacts d'alias.

## 4.5 Etude des impacts d'alias

Cette section consiste à vérifier s'il y a des impacts d'alias dans un programme. Tout d'abord, afin de réduire l'espace d'étude, je présente la condition nécessaire pour que les impacts d'alias apparaissent.

### 4.5.1 Condition d'apparition des impacts d'alias

Les impacts d'alias apparaissent s'il existe une nouvelle dépendance, c'est aussi la condition d'apparition de violation d'alias. C'est la condition nécessaire, mais pas suffisante. Elle est expliquée ci-dessous.

Supposons qu'il y a deux variables aliasées. Elles sont stockées dans deux séquences de stockage. Deux séquences de stockage sont dans la même section mais empilées l'une sur l'autre. Elles sont illustrées dans la figure ci-dessous:

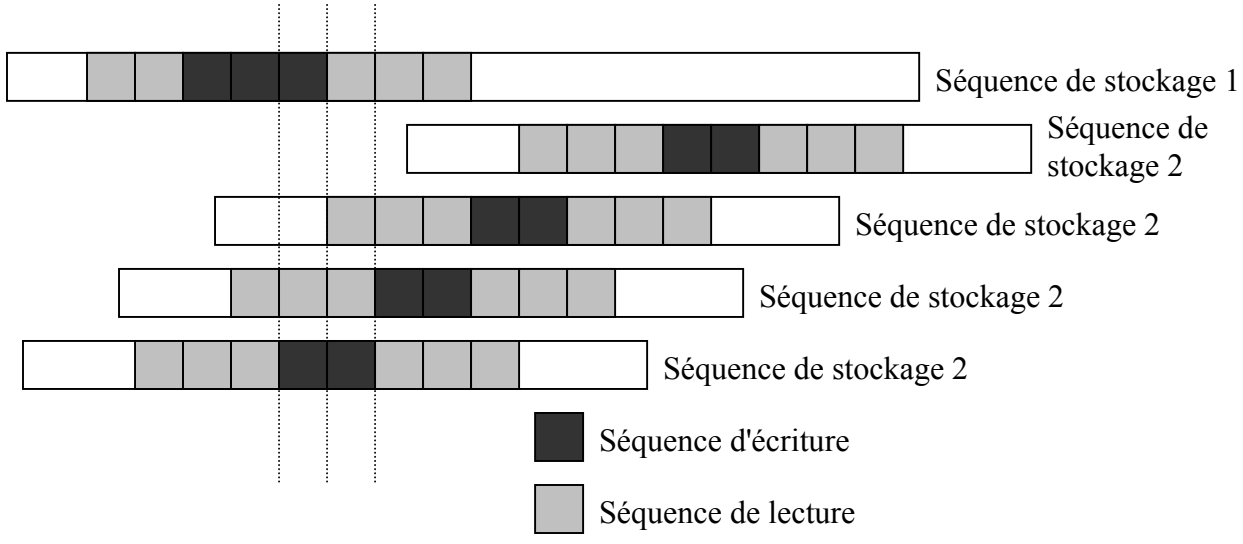


*Figure 17: Condition d'apparition d'impacts d'alias*

Notons que dans les cas ou existe une dépendance entre deux variables, au moins une variable doit être écrite.

Dans le premier cas: aucune séquence d'écriture ou de lecture de deux séquences de stockage

n'est empilée l'une sur l'autre. Il n'y a donc pas d'impacts d'alias dans ce cas.

Dans le deuxième cas: Seulement les séquences de lecture de deux séquences de stockage sont empilées l'une sur l'autre, c'est pourquoi il n'y a pas d'impacts d'alias dans ce cas.

Dans le troisième et quatrième cas: La séquence d'écriture de variable 2 est empilée par la séquence de lecture ou d'écriture de variable 1, et en conséquence, il existe une nouvelle dépendance qui peut créer un impact d'alias.

La condition d'apparition d'alias est la condition nécessaire, pas suffisante mais sur laquelle on peut se baser pour réduire l'espace d'étude. L'algorithme d'étude des impacts d'alias utilisant cette condition est présenté dans la section suivante.

### *4.5.2 Algorithme*

L'idée de cet algorithme est simple : parcourir toutes les associations alias pour trouver les alias, ensuite chercher les nouvelles dépendances créées par ces alias, et enfin tester si ces nouvelles dépendances sont redondantes avec les anciennes dépendances.

L'implémentation de cet algorithme n'est pas encore complète, elle ne traite que le problème d'impact d'alias entre deux variables scalaires. Cette implémentation ne considère que le nom de variable, pas son indice. Chaque fois qu'une instruction écrit sur une variable, on considère qu'elle écrit sur tout l'emplacement mémoire que la variable occupe. Dans le cas des variables tableaux, l'indice de variable doit être considéré pour chercher la nouvelle dépendance. Cette implémentation est à développer.

```
Fonction impact_check
    Obtenir le graphe de dépendance G
    Obtenir la liste des associations alias L
    Obtenir le statement du module MS
    Pour chaque association alias aa1 dans la liste L Faire
        Obtenir la variable v1 appartenant à aa1
        /* chercher alias entre deux paramètres */
        Pour chaque association alias aa2 dans le reste de la liste L Faire
            Obtenir la variable v2 appartenant à aa2
            Utiliser l'algorithme d'une phase existante pour tester s'il existe une violation
            d'alias entre v1 et v2 ou pas
            Si oui Faire
                Initialiser les listes stat_reads1, stat_writes1, stat_reads2, stat_writes2
                /* stat_reads1 est une liste stockant les statements qui écrivent v1 et
                ainsi de suit pour stat_writes1, stat_reads2, stat_writes2 */
                appeler check_new_arc_for_structured_statement(MS)
            FinSi
        FinPour
        /* chercher alias entre un paramètre et une variable common */
        Pour chaque variable v2 dans module Faire
```

```
                    Si v2 est une variable common Faire
                        Utiliser l'algorithme d'une phase existante pour tester s'il existe une
                        violation d'alias entre v1 et v2 ou pas
                        Si oui Faire
                            Initialiser les listes stat_reads1,2 et stat_writes1, 2
                            appeler check_new_arc_for_structured_statement(MS)
                        FinSi
                    FinSi
                FinPour
            FinPour
Fin
```

```
Fonction check_new_arc_for_structured_statement(statement s)
    Si s est un statement call Faire
        obtenir les effets du statement s
        Si s lit v1 Faire
            construire la nouvelle liste stat_reads1 contenant seulement s
            Pour chaque statement sw2 dans la liste stat_writes2 Faire
                chercher un chemin entre s et sw2 dans G
                Si il existe un chemin Faire
                    /* il y a un impact d'alias, nouvelle flow-dépendance créée */
                    imprimer la description d'impact
                FinSi
            FinPour
        AutreSi s écrit v1 Faire
            construire la nouvelle liste stat_writes1 contenant seulement s
            Pour chaque statement sr2 dans la liste stat_reads2 Faire
                chercher un chemin entre s et sr2 dans G
                Si il existe un chemin Faire
                    /* il y a un impact d'alias, nouvelle anti-dépendance créée */
                    imprimer la description d'impact
                FinSi
            FinPour
            Pour chaque statement sw2 dans la liste stat_writes2 Faire
                chercher un chemin entre s et sw2 dans G
                Si il existe un chemin Faire
                    /* il y a un impact d'alias, nouvelle output-dépendance créée */
                    imprimer la description d'impact
                FinSi
            FinPour
        FinSi
```

Faire aussi comme v1 pour variable v2

**AutreSi** s est un statement test **Faire**

sauvegarder les listes anciennes stat_reads1, stat_reads2, stat_writes1, stat_writes2

appeler check_new_arc_for_structured_statement(partie_vrai(s))

sauvegarder les nouvelles listes stat_reads1, stat_reads2, stat_writes1, stat_writes2

restaurer les listes anciennes stat_reads1, stat_reads2, stat_writes1, stat_writes2

appeler check_new_arc_for_structured_statement(partie_faux(s))

réunir deux ensembles des listes obtenues après chaque appel

**Sinon**

aller plus profondément pour obtenir le corps du statement s

appler check_new_arc_for_structured_statement(corps(s))

**FinSi**

## *4.6 Expérimentations*

Cette phase a été implémentée dans PIPS pour les cas d'impact d'alias entre les variables scalaires. Il reste encore le cas d'impact d'alias entre les variables tableaux à implémenter. La validation est effectuée avec plusieurs codes écrits par l'auteur et certains vrais codes de SPEC95.

Pour plus de détails sur la structure de données voir la partie *Annexe*.

Dans les deux exemples ci-dessous, l'entrée est un programme Fortran, la sortie est l'affichage des descriptions des impacts d'alias.

**ENTREE**

```
      PROGRAM IMPACT
      COMMON /COM/ T
      INTEGER A, B, C(5), D
      A = 0
      B = 1
C     no alias
      CALL FOO(A, B, C, D)
C     alias between two scalars
      CALL FOO(A, A, C, D)
      CALL FOO(A, B, C, A)
C     alias between one formal variable with one common variable
      CALL FOO(T, B, C, D)
C     alias between one scalars and one array
      CALL FOO(A, B, C, C(1))
C     no alias
      CALL FOO3(A, B)
C     alias between two arrays
      CALL FOO1(C(1), C(2))
      END

      SUBROUTINE FOO(X,Y,Z,ZZ)
      COMMON /COM/ T
      INTEGER X, Y, Z(5), ZZ
      X = X + 1
      T = Y
      Y = X
      X = T
      DO I=1, 5
```

```
         Z(I) = ZZ * Z(I)
      ENDDO
      END

      SUBROUTINE FOO1(X,Y)
      INTEGER X(5), Y(4)
      CALL FOO2(X(2), Y(2))
      END

      SUBROUTINE FOO2(X, Y)
      INTEGER X(4), Y(3)
      CALL FOO3(X(2),Y(1))
      END

      SUBROUTINE FOO3(X,Y)
      INTEGER X, Y
      X = 5
      Y = 6
      END
```

**SORTIE:** les impacts d'alias dans le programme

```
Impact alias at the call path (IMPACT:(0,5)) , between Y and X
New flow-dependence between
      X = X+1                                                          0001
and
      T = Y                                                            0002
Impact alias at the call path (IMPACT:(0,6)) , between ZZ and X
New flow-dependence between
      X = T                                                            0004
and
      Z(I) = ZZ*Z(I)                                                   0006
Impact alias at the call path (IMPACT:(0,7)) , between X and T
New anti-dependence between
      X = X+1                                                          0001
and
      T = Y                                                            0002
Impact alias at the call path (IMPACT:(0,7)) , between X and T
New output-dependence between
      X = X+1                                                          0001
and
      T = Y                                                            0002
Impact alias at the call path (FOO2:(0,2)) (FOO1:(0,2)) (IMPACT:(0,10))
, between X and Y
New output-dependence between
      X = 5                                                            0001
and
      Y = 6                                                            0002
```

**ENTREE**

```
      PROGRAM IMPACT
      COMMON /COM/ T
      INTEGER A, B
      A = 0
      B = 1
      CALL FOO(A,B)
      CALL FOO(A, A)
```

```
      END

      SUBROUTINE FOO(X,Y)
      COMMON /COM/ T
      X = 0
      IF (T.GT.0) THEN
         X = 0
      ELSE
         Y = 1
      END IF
      Y = 0
      END
```

**SORTIE:** les impacts d'alias dans le programme

```
Impact alias at the call path (IMPACT:(0,5)) , between Y and X
New output-dependence between
      X = 0                                                          0001
and
      Y = 1                                                          0004
Impact alias at the call path (IMPACT:(0,5)) , between Y and X
New output-dependence between
      X = 0                                                          0001
and
      Y = 0                                                          0005
Impact alias at the call path (IMPACT:(0,5)) , between Y and X
New output-dependence between
      X = 0                                                          0003
and
      Y = 0                                                          0005
```

# Conclusion

### *Résultats obtenus*

Dans le cadre du stage, j'ai implémenté trois phases manquantes dans PIPS.

La première phase est la phase du filtrage et de la visualisation du graphe de dépendance. Cette phase est nécessaire pour l'utilisateur afin d'analyser les dépendances de grands programmes compliqués. Elle permet de filtrer le graphe de dépendance sur certaines variables et de représenter le graphe de dépendance sous forme graphique. Elle a été bien validée par plusieurs tests.

La deuxième phase est la phase du filtrage et de visualisation des effets mémoire. Cette phase a pour l'objectif que l'utilisateur puisse savoir exactement dans quelle procédure et dans quel chemin d'appel des procédures certaines variables sont écrites ou lues et plus précisément l'instruction créant ces effets mémoire. Afin de faciliter la lecture de l'affichage des résultats, la visualisation des effets mémoire associés au graphe de flot de contrôle est ajoutée. Cette phase a été bien validée par plusieurs tests.

La dernière phase est la phase d'impact des alias sur le graphe de dépendance. Elle nous permet de savoir dans quel chemin d'appel des procédures il y a des influences sur la sémantique du programme et à cause de quelles variables. En implémentant cette phase nous pouvons détecter des erreurs d'un programme lorsqu'il est parallélisé. Cet apport est une modeste contribution au domaine très vaste de la parallélisation. Cette phase a été validée par plusieurs tests.

### *Problèmes restant*

Pour la phase d'impact des alias sur le graphe de dépendance, il reste encore le cas des variables tableaux. Jusqu'à ce moment, l'impact des alias a été bien étudié et son algorithme est bien construit pour les variables scalaires. Dans cet algorithme on ne considère que le nom de variable, pas son indice. Il reste à améliorer cet algorithme dans l'avenir en considérant les indices de variables et en les comparant pour trouver les vraies nouvelles dépendances.

### *Acquis à partir du stage*

C'était la première fois que je participe au développement d'un grand projet comme PIPS qui est développé depuis plusieurs années et qui se poursuivra. La participation à PIPS exige une méthode  pour pouvoir accomplir une tâche dans un délai défini.

En outre, pour entrer dans le domaine de la parallélisation, j'ai dû lire beaucoup de livres et des articles sur la compilation, l'optimisation, la dépendance, la transformation, ... Avec trois phases ajoutées dans PIPS, j'ai dû écrire environ 1100 lignes de codes C. Pour moi, le grand problème rencontré est l'insuffisance de documentation pour les librairies de PIPS, cela me force à apprendre de lire et de comprendre rapidement les codes sources de librairies de PIPS et des autres modules.

# Bibliographie


[BalMar79] A.Balfour et D.H. Marwick, *Programming in Standard Fortran 77,* 1979.

[AlKe87] R.Allen, K. Kennedy, *Automatic Translation of FORTRAN programs to Vector Form,* ACM Transactions on Programming Languages and Systems, Vol. 9

[Flynn66] M. Flynn, *Very High-Speed Computing Systems,*. In Proc, IEEE, vol.54, pp 1901-1909, 1966.

[HPF97] *High Performance Fortran Language Specification*, version 2.0, Jan. 1997.

[IJT91] F.Irigoin, P. Jouvelot, R. Triolet, *Semantical Interprocedural Prallelization : An Overview of the PIPS Project,* In Proceedings of the 1991 ACM International Conference on Supercomputing, Cologne, Germany, June 1991

[Kerr87] J. Kerridge, *OCCAM Programming: a Practical Approach,* Blackwell Scientific Publications, 1987.

[Kuck78] D. Kuck, *The Structure of computers and computation,* Vol.1, John Wiley and Sons, NewYork, 1978.

[MeRe89] M. Metcalf, J. Reid, *Fortran 8x Explained,* Oxford Science publications, 1989.

[Nga02] Nguyen Thi Viet Nga, *Verification des logiciels par analyses de programmes*, PhD Thesis 2002.

[Silber99] Georges-André Silber, *Parallélisation automatique par insertion de directives*, PhD Thesis 1999.

[Wolfe95] Michael Wolfe, *High Performance Compilers For Parallel Computing*, 1995

[Yang92] Yi-Qing Yang, *Minimal Data Dependence Abstractions for Loop Transformations,* 1992.

[Yang93] Yi-Qing Yang, *Tests des Dépendances et Transformations de Programme*, 1993.

[Youcef02] Youcef Bouchebaba, *Pavage pour une séquence de nids de boucles*, 2002.

# Annexe

### *Représentation interne de PIPS (Structure de données)*

Cette partie décrit la structure de données de PIPS mais seulement la partie utilisée dans le travail de l'auteur.

**graph**

*graph = vertices : vertex**

Le graphe de dépendance est stocké dans la structure *graph*, elle se compose d'une liste de vertex.

**vertex**

*vertex = vertex_label x successors : successor**

Un *vertex* représente un sommet de graphe, se compose d'une étiquette et une liste de successeurs associés à cette étiquette.

**successor**

*successor = arc_label x vertex*

Un *successor* représente un arc de graphe, se compose d'une étiquette et un vertex qui est indiqué par cet arc.

**statement**

Une instruction est représentée par le type statement.

*statement = label:entity x number:int x ordering:int x comments:string x instruction*

L'étiquette d'un statement et son commentaire sont stockés dans le champ *label* et *comments*. L'instruction du statement est décrite par le champ *instruction.* Le champ *number* n'est pas utilisé dans PIPS, le champ *ordering* est utilisé comme l'identité interne de statement.

**call**

Le type *call* représente les commandes de FORTRAN, des fonctions définies par les utilisateurs et des appels de sous-routines à la manière d'une pseudo-fonction pour les autres représentations. Des constantes, des opérateurs (+, -, .AND., .OR., …), des intrinsèques (SIN, MIN, MOD, …), des instructions de base (affectation, CALL, RETURN, …) comme des fonctions définies par les utilisateurs sont tous représentées par *call*

*call = function:entity x arguments:expression**

Le champ *function* pointe vers une entité qui associe avec la fonction appelée. A travers ce champ, on peut obtenir des informations sur la fonction (son nom, son type de données retour, etc). Un argument est vraiment une expression, la liste d'arguments passés est stockée dans une liste des expressions.

**expression**

Le type *expression* représente une expression de FORTRAN. Une expression peut être une référence (variable ou élément d'un tableau), une constante, un appel de fonction avec arguments, un intervalle d'une boucle, … ou même une expression de sous expressions.

Elle est définie comme suit:

*expression = syntax x normalized*

*syntax = reference x range x call*

*reference = variable:entity x indices:expression**

Dans la définition de *reference, indices* est une liste d'expressions pour représenter des dimensions de tableau. Cette liste est vide si *reference* est une variable scalaire.

**entity**

Un *entity* est utilisé pour représenter tous les objets nommés dans un programme Fortran, comme module (fonction, sous-routine, programme), une variable, un opérateur, un intrinsèque, une constante, une étiquette, etc.

*entity = name: string x type x storage x initial: value*

Le champ *name* contient le nom de l'objet dans le programme source tandis que le champ *type* indique le type de l'objet (module, variable, …). *Storage* définit l'allocation mémoire et *initial* (objet de type *value*) contient la valeur initiale de l'objet.