No d'ordre

THÈSE de DOCTORAT de L'UNIVERSITÉ PARIS VI

Spécialité :

Systèmes Informatiques

présentée

par Madame ***Yi-Qing YANG***

pour obtenir le titre de **DOCTEUR DE L'UNIVERSITÉ PARIS VI**

Sujet de la Thèse :

Tests des Dépendances et Transformations de Programme

soutenue le *15 Novembre 1993*

devant le jury composé de:

Madame Corinne Ancourt	Examinateur
Monsieur Paul Feautrier	Examinateur
Monsieur Claude Girault	Président
Monsieur Francois Irigoin	Examinateur
Monsieur Yves Robert	Rapporteur
Monsieur Francois Thomasset	Rapporteur

ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

Résumé

La parallélisation d'un programme séquentiel comporte plusieurs étapes : le calcul des dépendances, leur représentation et l'utilisation de cette représentation pour l'application des transformations de programme permettant d'obtenir un ordonnancement *parallèle* des instructions du programme. Le succès de la parallélisation dépend de la précision du test de dépendances et des représentations utilisés pour ces dépendances.

Nous présentons et comparons, dans cette thèse, différents algorithmes de test de dépendances et différentes abstractions de ces dépendances. L'algorithme du paralléliseur *PIPS* est basé sur un test de faisabilité approximatif utilisant l'algorithme de *Fourier-Motzkin*. Nos expériences montrent que, dans la pratique, il est suffisamment précis pour traiter des systèmes de dépendances et que sa complexité pratique est polynômiale.

Les différentes abstractions des dépendances ont des précisions différentes. Pour effectuer légalement une transformation, plusieurs abstractions sont *admissibles*, c'est à dire contiennent *suffisamment* d'information pour savoir si la transformation peut être appliquée légalement. L'abstraction *minimale* est celle qui contient l'information nécessaire minimale appropriée à la transformation. Nous avons identifié l'abstraction admissible minimale appropriée aux transformations de programme classiques : *inversion de boucle, permutation de boucles, transformations unimodulaires, partitionnement et parallélisation.*

Le cône de dépendance, qui est l'abstraction admissible et minimale pour l'application de toute transformation unimodulaire, contient aussi suffisamment d'information pour obtenir l'ensemble des *ordonnancements* linéaires valides mono- et multi-dimensionnels, identique à celui calculé à partir de l'abstraction des vecteurs de distance de dépendance.

Mots cles : parallélisation automatique, dépendance, test de dépendance, abstraction de dépendance, transformation de programme, réordonnancement.

Abstract

The parallelization of sequential programs requires several stages : analysis of dependence relations, representation of these dependences and application of transformations using this representation to find a *parallel* schedule for the program instructions. The success of parallelization depends on the precision of the dependences test and dependence representation used.

In this thesis, we present and compare different dependence test algorithms and different data dependence abstractions. The algorithm of the PIPS parallelizer is based on a approximate feasibility test using *Fourier-Motzkin* elimination. Our experiments show that, in practice, it is accurate enough for treating dependences systems, and that its practical complexity is polynomial.

Different dependence abstractions have different precision. For deciding whether a transformation is legal, several abstractions are *admissible*, meaning they contain *enough* information for knowing if this transformation is legal. The *minimal* abstraction contains *necessary minimal* information for this transformation. We have identified the admissible minimal abstractions for classical program transformations : *loop reversal, loop permutation, unimodular transformations, partitioning and parallelization.*

The dependence cone, which is the admissible minimal abstraction for the application of all unimodular transformations, also contains sufficient information for obtaining the mono- and multi-dimensional *valid linear scheduling* set, identical to the one computed from the abstraction of dependence distance vectors.

Remerciements

Je tiens tout d'abord à remercier le Professeur Michel Lenci, ancien Directeur du Centre de Recherche en Informatique de l'Ecole des Mines de Paris, qui m'a accueillie dans son laboratoire et a toujours porté une grande attention à mes travaux.

Je veux remercier également le Professeur Robert Mahl, Directeur du Centre de Recherche en Informatique, qui m'a conseillée et a consacré une partie de son temps à la relecture de ma thèse.

J'adresse mes vifs remerciements aux membres du jury:

- *au Professeur Claude Girault qui me fait l'honneur de présider le jury.*
- *à Francois Thomasset et au Professeur Yves Robert qui ont accepté d'être rapporteur.*
- *au Professeur Paul Feautrier qui, en tant que directeur de cette thèse, a consacré du temps à la lecture de mon manuscrit.*
- *à Francois Irigoin, Directeur adjoint du centre, qui m'a proposé un sujet intéressant, a dirigé cette thèse, a lu et approfondi les rédactions intermédiaires et a accepté de faire partie des membres du jury. Ses suggestions et critiques ont permis de faire avancer ce travail.*
- *à Corinne Ancourt, ma collègue, qui a corrigé consciencieusement les premières versions du manuscrit. Elle a apporté à cette thèse non seulement des corrections aux fautes de Français mais aussi des critiques et des remarques importantes.*

Je remercie également Pierre Jouvelot pour sa disponibilité et Fabien Coelho qui a participé à la relecture.

Enfin, je veux remercier toutes les personnes du CRI, qui ont contribué de près ou de loin au bon déroulement de ce travail par leurs encouragements, à J. Altimira pour son aide amicale et à A. Pech qui m'a aidée lors des recherches bibliographiques.

Table des matières

Table des figures

Liste des tableaux

Introduction

De nombreuses applications scientifiques des domaines de météorologie, défense, énergie nucléaire et sismique, manipulent un grand nombre de données. Pour rester compétitives et prendre en considération le nombre de données toujours croissant, ces applications doivent être traitées en *parallèle*.

La parallélisation d'un programme séquentiel comporte trois phases essentielles: le test de dépendance, les transformations du programme (permettant une détection plus facile du parallélisme implicite) et enfin la parallélisation. Le premier chapitre de cette thèse introduit les méthodes générales de parallélisation et vectorisation des programmes.

Le test de dépendance est une des phases les plus importantes pour la détection et l'exploitation du parallélisme implicite des programmes. Sa précision et son efficacité conditionnent directement le succès de la phase de parallélisation. Pour effectuer un test de dépendance, de nombreux algorithmes, exacts ou approximatifs, de complexité polynômiale ou exponentielle, ont été proposés. Les tests approximatifs approximent l'ensemble des solutions et ne testent que l'existence (ou l'inexistence) d'une telle solution. Bien que de nombreux tests de dépendances proposés soient approximatifs, très peu d'études ont porté sur l'évaluation de la complexité et de l'exactitude de tels tests sur les programmes réels. L'algorithme du paralléliseur *PIPS* est basé sur un test de faisabilité approximatif utilisant l'algorithme de *Fourier-Motzkin*. Le deuxième chapitre de cette thèse expose les résultats de l'évaluation de sa complexité et de son exactitude.

Afin de paralléliser le programme, ce dernier doit être restructuré et transformé. Une transformation peut être appliquée au programme si toutes les dépendances entre les instructions du programme sont respectées. Une représentation exacte ou approximative caractérisant ces dépendances est donc nécessaire. De

nombreuses abstractions ont été proposées: les itérations de dépendance, les vecteurs de distance de dépendance, le cône de dépendance, les vecteurs de direction de dépendance et les profondeurs de dépendance. Ces différentes abstractions ont des précisions et des coûts de calcul et stockage différents. Après une description de ces différentes abstractions, une comparaison de leur précision est présentée dans le troisième chapitre.

La précision des abstractions des dépendances est importante lors de l'application des transformations de programme car certaines transformations pourront être déclarées "illégales" pour une abstraction et pas pour une autre (contenant suffisamment d'information sur les dépendances pour pouvoir tester si elle peut être appliquée sans contrarier les dépendances). Les abstractions admissibles pour chacune des transformations sont parfois multiples. La précision de chacune des abstractions étant différente, le choix d'une abstraction des dépendances appropriée à une transformation est important. Les résultats de l'étude des relations entre abstractions de dépendances et transformations de boucles sont exposés au chapitre 4.

La parallélisation d'un nid de boucles peut être vue comme un problème de réordonnancement des itérations du nid de boucles de manière à mettre en évidence des boucles parallèles. Des méthodes ont donc été proposées pour calculer un ordonnancement linéaire des itérations des boucles. Elles utilisent les vecteurs de distance de dépendance comme abstraction des dépendances. Le cône de dépendance est une abstraction des dépendances qui approxime l'ensemble des vecteurs de distance de dépendance et est plus compact. Le dernier chapitre de cette thèse est dédié à l'étude des ordonnancements valides que l'on peut calculer à partir de cette représentation.

Chapitre 1

État de l'art

De nombreuses applications, de domaines scientifiques différents comme la météorologie, la défense, l'énergie nucléaire et la sismique, manipulent un grand nombre de données. Pour rester compétitives et prendre en considération le nombre de données toujours croissant, ces applications doivent être traitées en *parallèle*. De nombreux modèles d'architectures parallèles (i.e. MIMD, SIMD, VLIW) ont été proposés pour exploiter le parallélisme de ces applications à différents niveaux (i.e. tâche, programme, sous-routine, boucle, instruction). Nous présentons brièvement les principaux types 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 soit explicitement par les programmeurs, dans les programmes à l'aide des primitives d'un langage parallèle, soit implicitement. S'il est implicite, le parallélisme devra être détecté par les compilateurs/paralléliseurs. Dans la deuxième section de ce chapitre, nous présentons quelques langages de programmation parallèle.

Pour conserver les bibliothèques scientifiques importantes déjà existantes, et pour que les programmeurs puissent continuer à écrire des programmes séquentiels, des outils d'aide à la parallélisation sont nécessaires. Dans la deuxième partie de ce chapitre, nous décrivons les techniques classiques, utilisées par ces outils, telles que la détection des dépendances, les transformations de programme et les méthodes de la vectorisation/parallélisation.

1.1 Architectures parallèles

Comme leur nom l'indique, les machines parallèles sont des ordinateurs pouvant exécuter plusieurs instructions simultanément. Elles possèdent soit plusieurs processeurs, soit un processeur avec plusieurs unités de traitement, soit une unité pipelinée, soit encore une combinaison des trois catégories précédentes. Il faut noter que tous les microprocesseurs récents comme les *superscalaires*: SuperSPARC, RS6000, MC88100 et les *superpipelines*, comme le R4000, qui 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 [Flyn66] 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 est sans intérêt pratique et celle des SISD correspond aux machines séquentielles. Nous nous intéressons donc dans cette thèse aux deux catégories SIMD et MIMD.

- **SIMD** (un seul flux d'instructions, plusieurs flux de données)
  C'est le type des architectures massivement parallèles dont la CM-2 est le meilleur exemple. 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** (plusieurs flux d'instructions et de données)
  C'est l'architecture du 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 2, 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. Fabriquer des programmes largement parallèles est la clé nécessaire à l'obtention de bonnes performances pour ces architectures. Dans les deux sections suivantes, nous présentons deux approches 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 proposés. Nous présentons brièvement trois exemples de langage parallèle qui ont des modèles de programmation différents et qui sont dédiés à trois types différents d'architecture parallèle : 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, elles 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 naturel-

lement 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 nombres de processus.

Voici un exemple de deux processus communiquants :

```
PROC p1(CHAN com)
     VAR x :
     SEQ
     ...
      x:=...
      comm! x:
     ...

PROC p2(CHAN com)
     VAR y:
     SEQ
     ...
     comm? 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 90

Le FORTRAN 90 est un langage conçu pour les machines vectorielles. Il est tout à fait compatible avec le FORTRAN 77, mais possède des opérations et des fonctions plus riches. La particularité la plus connue du FORTRAN 90 est le *traitement des tableaux en parallèle* [MeRe89].

Les opérations vectorielles principales sont :

- l'affectation de section de tableau,

  Par exemple,

  ```
  REAL DIMENTION(:,:):: 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 les expressions et les opérations vectorielles du langage.

1.2.3 HPF

HPF (*High Performance Fortran*) est un langage de programmation data-parallèle qui est en cours de spécification aux Etats-Unis [HPF92]. 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 performances optimales sur les machines MIMD et SIMD à mémoire non uniforme tout en préservant la portabilité. HPF a étendu Fortran sous plusieurs aspects : distribution des données, instructions parallèles, intrinsèques et librairies, etc.

Afin de permettre d'expliciter des 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 :

```
FORALL (I=2:N-1, J=2:N-1)
      A(I,J) = A(I,J-1)+A(I,J+1)
      B(I,J) = A(I,J)
END FORALL
```

  Ceci est équivalent à deux affectations vectorielles :

```
FORALL (I=2:N-1, J=2:N-1)  A(I,J) = A(I,J-1)+A(I,J+1)
FORALL (I=2:N-1, J=2:N-1)  B(I,J) = A(I,J)
```

- Exemple 2 :

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

  La directive INDEPENDENT affirme que les 100 itérations de la boucle `I` sont indépendantes.

  Le premier compilateur de HPF n'est pas encore mis à disposition, il est prévu pour la fin 1993.

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, et que le programme doit être transformé pour spécifier le parallélisme maximum.

L'utilisation des outils de parallélisation ou de vectorisation automatique devient dès lors nécessaire.

1.3 Vectorisation et parallélisation automatique

La deuxième approche consiste donc à paralléliser automatiquement les programmes séquentiels écrits en langages impératifs (ex. Fortran 77, C) grâce aux paralléliseurs. C'est le paralléliseur qui se charge de détecter le parallélisme implicite contenu dans les programmes et de l'exploiter. Cette approche est intéressante car elle permet aux programmeurs de continuer à écrire les programmes séquentiels en utilisant le langage qu'ils connaissent bien, le compilateur prenant en charge la parallélisation de ces programmes. De plus, elle permet de conserver les bibliothèques scientifiques importantes utilisées sur les machines parallèles sans les réecrire.

Les deux approches, vectorisation et parallélisation, visent respectivement les deux types de machines parallèles, les machines vectorielles (SIMD) et les multiprocesseurs (MIMD).

Nous nous intéressons dans cette thèse tout particulièrement à la parallélisation du langage Fortran, puisque c'est le langage le plus utilisé dans le monde scientifique.

Dans cette section, nous commençons par présenter trois types de parallélisme implicite. Ensuite, nous nous concentrons sur l'exploitation du parallélisme contenu dans les boucles. C'est, en effet, dans les nid de boucles que se trouve le plus fort potentiel de parallélisme implicite exploitable. Le parallélisme de contrôle a été estimé par de nombreux auteurs à 2 ou 3 instructions pouvant s'exécuter en parallèle. De plus, le projet PTRAN a montré qu'il n'y avait pas grand chose à faire en dehors des boucles.

Nous présentons ensuite les phases essentielles et les techniques utilisées pour réaliser la parallélisation ou la vectorisation. Il s'agit
1) du test de dépendance permettant de détecter les contraintes d'ordonnancement entre instructions,
2) de l'utilisation des transformations de programmes et

3) de la parallélisation/vectorisation explicitant le parallélisme.

1.3.1 Parallélisme implicite

Nous présentons ici trois types de parallélisme implicite : le parallélisme de gros grain, de moyen grain et de grain fin.

Parallélisme de gros grain

Dans les programmes d'analyse numérique, modules et boucles représentent des parties relativement importantes du programme. L'exécution de tels blocs d'instructions peut être parallèle si les calculs intervenant dans ces blocs sont indépendants. Ce type de parallélisme est considéré comme du parallélisme de gros-grain. Par exemple, dans le langage **OCCAM**, la primitive **PAR** permet d'exprimer ce type de parallélisme.

Illustrons deux situations : l'une où les procédures sont indépendantes et peuvent être exécutées en parallèle, l'autre où ce sont les boucles qui sont indépendantes et peuvent être exécutées en parallèle.

1. exemple 1 :

```
      PROGRAM P1        SUBROUTINE SUB1(A,n)   SUBROUTINE SUB2(A,n)
      PAR
bloc1:CALL SUB1(A,n)     DO I = 1, 2*n-1, 2     DO I = 2, 2*n, 2
bloc2:CALL SUB2(A,n)        A(I) = 0               A(I) = 1
      ENDPAR              ENDDO                  ENDDO
      END                END                    END
```

Ce programme contient deux appels à une procedure : `CALL SUB1`, `CALL SUB2`. SUB1 affecte aux éléments impairs du tableau `A` la valeur 0, et `SUB2` affecte la valeur entière 1 aux éléments pairs du tableau `A`. Il est clair que ces deux appels sont indépendants et que le `bloc1` et le `bloc2` peuvent être exécutés en parallèle.

2. exemple 2 :

```
          PROGRAM P2
             PAR
bloc1:         DO I = 1, 2*n-1, 2
                  A(I) = 0
               ENDDO
bloc2:         DO I = 2, 2*n, 2
                  A(I) = 1
               ENDDO
             ENDPAR
          END
```

Le programme `P2` est composé de deux boucles successives. En fait, `P2` fournit le même résultat que `P1` (les appels aux sous-routines sont remplacées par leur corps d'instructions). Les exécutions des blocs `bloc1` et `bloc2` sont pour la même raison indépendantes, puisqu'ils peuvent être exécutés en parallèle.

Parallélisme de moyen grain

Nous appelons le parallélisme de grain moyen le parallélisme correspondant à l'exécution parallèle des itérations d'une boucle.

La majorité des paralléliseurs exploitent ce type de parallélisme. Il est explicité par trois types de structures parallèles :

- l'instruction vectorielle comme celle du Fortran 90,
  ex. `A(1:N) = B(1:N) * C(1:N)` (chaque élément `[1:N]` du tableau `A` peut être affecté en parallèle);

- la boucle `DOALL` qui signifie que toutes les itérations de la boucle peuvent être exécutées parallèlement, sans synchronisation.

- la boucle `DOACROSS` où toutes les itérations peuvent être exécutées parallèlement avec quelques synchronisations ou délais [Padu79] [Cytr84].

Prenons un exemple,

```
        DO I = 1, 5
S1:        A(I) = A(I) + 1
S2:        B(I) = B(I) + A(I)
S3:        C(I) = C(I) + B(I)
        ENDDO
```

FIG. 1.1 - *Exemple 1.1*

Ces instances sont représentées dans la figure 1.2. Les relations de dépendance entre les instances des instructions sont représentées par une flèche.

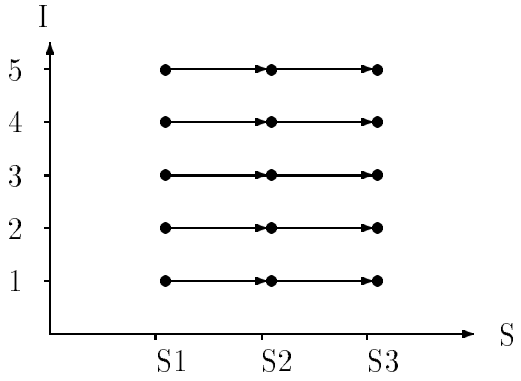


FIG. 1.2 - *Espace des instances de l'exemple 1.1*

On peut constater qu'il n'y a pas de dépendance entre les itérations du corps de boucles. En explicitant le parallélisme vertical par un `DOALL`, l'exemple devient :

```
        DOALL I = 1, 5
S1:        A(I) = A(I) + 1
S2:        B(I) = B(I) + A(I)
S3:        C(I) = C(I) + B(I)
        ENDDOALL
```

Une boucle peut contenir à la fois du parallélisme de gros grain et du parallélisme de moyen grain. Prenons un exemple,

```
        DO I = 1, 5
S1:       DO J = 1, N
             A(I,J) = A(I-1,J) + 1
          ENDDO
S2:       DO K = 1, M
             B(I,K) = B(I-1, K) + 1
          ENDDO
        ENDDO
```

FIG. 1.3 - *Exemple 1.2*

`S1` et `S2` sont deux instructions (blocs de boucle) du corps de la boucle `I`. Les relations de dépendance entre les instances de `S1` et `S2` sont illustrées dans la figure 1.4. Il y a des dépendances entre les différentes itérations de la boucle `I`. Mais il y a du parallélisme de moyen grain entre les itérations de la boucle `J` et celles de la boucle `K`, que l'on explicite par un DOALL et du parallélisme gros grain entre `S1(I)` et `S2(I)`, que l'on explicite par la primitive PAR.

Le code devient :

```
        DO I = 1, 5
           PAR
S1:           DOALL J = 1, N
                 A(I,J) = A(I-1,J) + 1
              ENDDOALL
S2:           DOALL K = 1, M
                 B(I,K) = B(I-1, K) + 1
              ENDDOALL
           ENDPAR
        ENDDO
```

Le parallélisme de moyen grain est un cas particulier du parallélisme de gros grain, les blocs sont identiques. Ces deux types de parallélisme sont exploités pour les machines parallèles SIMD vectorielles ou MIMD.

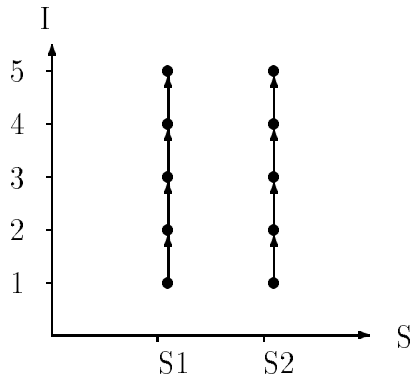


FIG. 1.4 - *Espace des instances de l'exemple 1.2*

Parallélisme de grain fin

On définit le bloc de base comme étant un bloc d'affectations n'ayant qu'une seule entrée et qu'une seule sortie. Plusieurs instructions indépendantes d'un même bloc peuvent s'exécuter en parallèle même si le taux de parallélisme supplémentaire obtenu n'est pas très important par rapport au parallélisme de grain moyen. Ce parallélisme au niveau des instructions est appelé le parallélisme de grain fin. Ce type de parallélisme est exploité pour les machines parallèles de type VLIW (Very Long Instruction Word), superscalaire et pipeline.

Une approche efficace, *Software Pipelining*, permettant d'exploiter le parallélisme de grain fin dans une boucle DO a été proposée dans [WaEi93].

Puisque, dans la majorité des programmes Fortran, la majeure part du temps d'exécution est consommée par les boucles [Kuck84], l'exploitation du parallélisme dans les boucles est cruciale. Les vectoriseurs et les paralléliseurs actuels s'attachent essentiellement à trouver les boucles du programme qui peuvent être vectorisées ou parallélisées. Dans cette thèse, nous nous intéressons donc tout particulièrement au parallélisme de gros ou moyen grain.

1.3.2 Détection des contraintes d'ordonnancement

Dans cette section, nous introduisons brièvement la première phase de la parallélisation : la détection des contraintes d'ordonnancement [Bane88] [Wolf89] [ZiCh90]. Pour cela certains concepts comme **dépendance**, **test de dépendance, graphe de dépendances, DDV, DC,** etc vont être introduits.

Cette phase doit analyser les relations de dépendance existant entre les instructions du programme.

Il existe deux types de relation de dépendance dans un programme : les dépendances de contrôle et les dépendances de données. Une dépendance de contrôle, respectivement de données, représente l'existence d'une contrainte d'ordonnancement sur le flux de contrôle, respectivement sur le flux de données. Toutes les dépendances de contrôle ou de données d'un programme sont généralement représentées par un graphe orienté qui s'appelle *le Graphe de Dépendances*. Puisque les dépendances de contrôle peuvent être traitées de la même manière que les dépendances de données, par un compilateur [FeOW87], nous n'étudions que les dépendances de données dans cette thèse.

L'algorithme de test des dépendances joue un rôle très important car il permet de savoir s'il existe une dépendance entre deux instructions référençant la même variable (scalaire ou tableau) et donc de savoir si les instructions sont parallélisables.

Test de dépendances des données

Rappelons simplement qu'une instruction $S2$ **dépend de** l'instruction $S1$ si $S1$ s'exécute avant $S2$ et si l'une des trois conditions suivantes[1] est vérifiée :

1. $S2$ lit la valeur d'une variable (ou référence du tableau) qui a été modifiée par $S1$; il y a une *flow-dependence* de $S1$ vers $S2$

2. $S2$ modifie une variable (ou référence du tableau) qui a été lue par $S1$; il y a une *anti-dependence* de $S1$ vers $S2$

3. $S2$ modifie une variable (ou référence du tableau) qui a été modifiée par $S1$; il y a une *output-dependence* de $S1$ vers $S2$

Pour des raisons de simplicité, nous prenons le cas de deux boucles imbriquées comme modèle.

1. un quatrième type de dépendance *input-dependence* a été ajouté pour l'optimisation des accès mémoire

```
      DO I = l1,  u1
        DO J = l2,  u2
S1            X (f1(I,J), f2(I,J)) = ...
S2                        ... = X (g1(I,J), g2(I,J))
      ENDDO
      ENDDO
```

On définit $S\ (i,j)$ l'instance de l'instruction S à l'itération $(I,J) = (i,j)$. L'exécution d'une instance $S\ (i,j)$ précède celle de l'instance $S'\ (i',j')$ si et seulement si

- $(i,j) \ll (i',j')$ (où $\ll$ défini l'ordre lexicographique)
- ou bien si S est avant S' textuellement et $(i,j) = (i',j')$.

L'ensemble des références de X modifiées par l'instruction $S1$ que l'on note $OUT(S1)$ est défini par $OUT(S1) = \{\ X\ (f_1(I,J),\ f_2(I,J))$ où $I \in [l_1,u_1], J \in [l_2,u_2]\ \}$. L'ensemble des références de X lues par l'instruction $S2$ noté $IN(S2)$ vaut $IN(S2) = \{X\ (g_1(I,J),\ g_2(I,J))$ où $I \in [l_1,u_1], J \in [l_2,u_2]\}$.

D'après la définition, $S2$ dépend de $S1$ par une flow dépendance si et seulement s'il existe des entiers i_1, i_2, j_1, j_2 tels que :

$$\begin{cases} f_1\ (i_1,j_1) = g_1\ (i_2,j_2) \\ f_2\ (i_1,j_1) = g_2\ (i_2,j_2) \\ l_1 \le i_1, i_2 \le u_1 \\ l_2 \le j_1, j_2 \le u_2 \\ (i_1,j_1) \ll (i_2,j_2) \end{cases}$$

Pour déterminer les dépendances, il faut rechercher l'existence de solutions entières au système d'équations et d'inéquations précédent. Obtenir une réponse exacte est équivalent à un problème de **programmation en nombres entiers** qui est NP-complet. Les algorithmes classiques de programmation en nombres entiers sont très coûteux. On est donc amené à rechercher des méthodes plus simples et plus efficaces.

Un grand nombre de tests, exacts ou approximatifs, ont été proposés : des tests exacts comme *PIP* [Feau88], *FAS*^{3}*T* [BePT90] et *Omega* [Pugh92]; des tests

vérifiant l'existence de solutions entières pour une ou toutes les équations du système comme le *PGCD* [Bane76] et le *PGCD généralisé* [Bane88]; des tests vérifiant l'existence de solutions réelles dans le système d'inégalités défini par le domaine d'itération comme le test de *Banerjee* [Bane76] [Bane79] [Bane88] et le test de *Fourier-Motzkin* [DaEa73] [Duff74]. Ces tests sont détaillés et comparés dans le chapitre suivant.

Graphe de dépendances des données

Les relations de dépendance entre toutes les instructions d'une boucle (ou un programme) sont exprimées par le graphe de dépendances qui est un graphe orienté. Soit $G(V, E)$ ce graphe, $V = \{S_1,\ S_2,\ \ldots\ S_n\}$ est la liste des sommets du graphe correspondant aux instructions du programme et $E = \{e_{ij} = (S_i,\ S_j) \mid S_i,\ S_j\ \in V\}$ est la liste des arcs exprimant les relations de dépendance entre les instructions du programme. On a choisi une structure de graphe pour représenter les relations de dépendance des instructions du programme parce que la dépendance est une relation d'ordre. La présence d'un cycle dans le graphe entre deux instructions S_1 et S_2 indique l'existence de contraintes d'ordonnancement entre certaines itérations de S_1 et certaines itérations de S_2. Il empèche l'exécution en parallèle de ces instructions. La détection facile et rapide des cycles du graphe de dépendances est indispensable pour la vectorisation et la parallélisation.

Prenons l'exemple suivant (l'exemple 1.3).

```
        DO I = 1, N
           DO J = 1, M
S1:            A(I+1,J) = A(I+1,J) + B(I,J-1)
S2:            B(I,J) = B(I,J) + A(I,J)
S3:            C(I,J) = A(I+1,J-1) + B(I-1,J)
           ENDDO
        ENDDO
```

FIG. 1.5 - *Exemple 1.3*

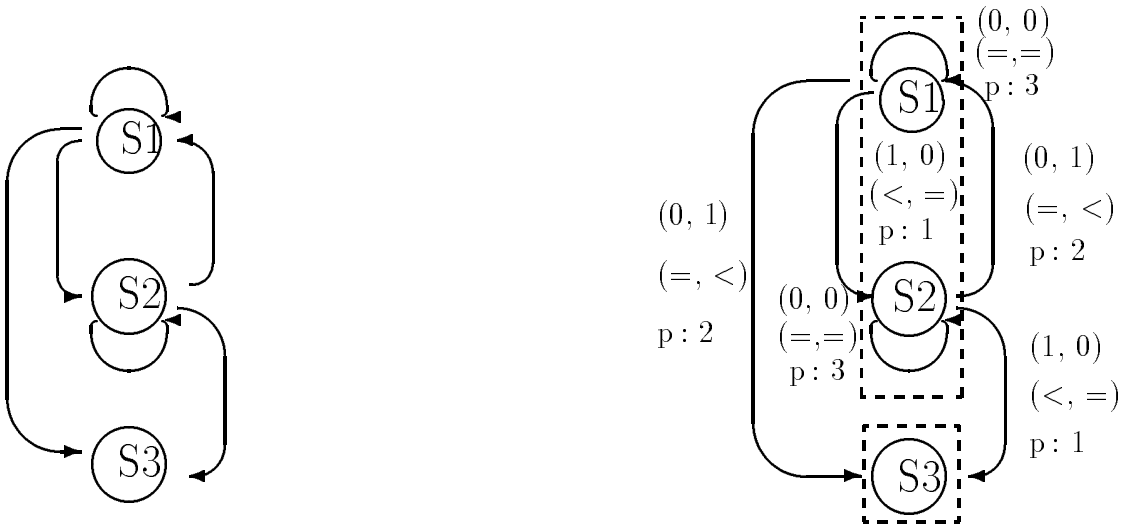


FIG. 1.6 - *Graphe de dépendances de l'exemple 1.3*

La partie gauche de la figure 1.6 représente le graphe de dépendances de l'exemple 1.3. Il y a six dépendances internes au nid de boucles dont quatre représentent des dépendances inter-itérations. Il y a trois cycles dont deux sont dûs à la lecture et à l'écriture d'une même variable au cours d'une même itération par une même instruction. $S3$ ne dépend pas transitivement d'elle même, elle peut donc être exécutée en parallèle après distribution[2]. $S1$ et $S2$ appartenant à un cycle doivent être exécutées sequentiellement.

Un graphe de dépendances ne comportant aucune information sur la nature des dépendances ne peut pas être utilisé pour effectuer des transformations de programme et des optimisations. On associe donc à chaque arc des informations qui caractérisent cette relation de dépendance.

On peut représenter une dépendance entre deux instructions $S1$ ($\vec{i}$) et $S2$ ($\vec{i'}$) par le vecteur de distance de dépendance $\vec{d}$, défini par la différence des deux vecteurs d'itérations $\vec{d} = \vec{i'} - \vec{i}$.

Si chaque élément de $\vec{d}$ est constant, on peut utiliser ce vecteur constant, représentant la **Distance de dépendance (D)**, comme information. Cependant $\vec{d}$ n'est pas toujours constant, et représente parfois une information trop riche. Certaines transformations, telle que l'échange de boucles, n'ont pas besoin de

2. La distribution de boucles est une transformation qui distribue les instructions par blocs (voir la section suivante)

cette information pour pouvoir être effectuées sans modifier la sémantique du programme [Wolf89].

Le **data Dependence Direction Vector (DDV)** est largement utilisé comme information de dépendance résumant le vecteur de dépendance par le signe de ses éléments. Il est de dimension égale au nombre de boucles imbriquées et ses composantes prennent leurs valeurs dans
$\Psi = (\psi_1, \psi_2, ..., \psi_d)$ où $\psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$ [Wolf89]. On dit que $S1$ $(i_1, i_2, ..., i_n)$ dépend de $S2$ $(i'_1, i'_2, ..., i'_n)$ avec le *ddv* Ψ si $\forall k (1 \leq k \leq n),\ i_k \psi_k i'_k$.

Lorsque les composantes du *ddv* sont toutes "=", la dépendance est interne à une itération et est appelée *loop independent* sinon elle a lieu entre deux itérations différentes et est appellée *loop carried* [AlKe87].

La **Profondeur d'une dépendance** est une des informations résumant le vecteur de direction de dépendance. Lorsque les `n-1` premières composantes du ddv sont "=" et la `n`-ième composante est <, par exemple (=, =, ..., =, <, ...), la dépendance est alors de **profondeur n** [AlKe87]. Cela veut dire que le signe de la `n`-ième composante du vecteur de dépendance est *positif*, contrairement à ce qui pourrait faire croire le symbole <.

Le graphe des dépendances de l'exemple 1.3 affiné par *le vecteur de distance, le ddv* et *la profondeur de dépendance* est illustré dans la partie droite de la figure 1.6.

Le **Cône de Dépendances (DC)** [IrTr87] est une autre manière intéressante de préciser les dépendances. Il caractérise l'ensemble des distances de dépendance (fini ou infini) par un polyèdre convexe minimal (ou l'enveloppe convexe) recouvrant toutes les distances. Il peut aussi être défini comme la fermeture transitive du vecteur de distance. Cette approximation est plus précise que le DDV qui est un cône particulier représentant soit un quart soit une moitié d'espace. Lorsque DC est un polyèdre convexe, il peut être représenté sous forme d'un système générateur [Schr86] composé de trois ensembles de vecteurs : sommets, rayons, droites. L'ensemble des vecteurs de distance de dépendance peut être généré précisément par des combinaisons linéaires de ces trois ensembles. Le DC d'un corps de boucles peut se caractériser par un seul cône, la fermeture transitive de toutes les relations de dépendance pour toutes les paires d'instructions et de références.

Il est particulièrement intéressant pour certaines transformations telles que les transformations de réordonnancement globale d'itérations, bien que l'enveloppe convexe fasse perdre des informations utiles pour d'autres transformations. Le réordonnancement des itérations par la méthode hyperplane déduit du DC est equivalent à celui déduit de l'ensemble des distances D [Irig88a].

1.3.3 Transformations

Expliciter le parallélisme d'un programme nécessite souvent des transformations de programme qui doivent conserver toutes les dépendances. Dans cette section, après avoir presenté les analyses et transformations de programme standards, nous présentons les transformations liées à la parallélisation/vectorisation.

Analyses et transformations standards

La plupart des paralléliseurs effectuent une phase de transformation pour faciliter le test de dépendance. Nous citons maintenant certaines de ces transformations.

- **Normalisation de boucle**

  Une boucle dont la borne inférieure et le pas sont égaux à 1 est considérée comme une boucle normalisée. La normalisation d'une boucle correspond à la transformation suivante :

  DO $I = e_1, e_2, e_3$ Corps (I) Enddo $\Longrightarrow$ DO $I' = 1, e_2', 1$ Corps (I') Enddo

  où e_1, e_2, e_3 sont des expressions des bornes et du pas de la boucle initiale. La nouvelle borne supérieure e_2' vaut $(e_2 - e_1 + e_3)/e_3)$. Le `Corps (`I'`)` est déduit du `Corps (`I`)` en remplacant I par l'expression $(e_1 + (I' - 1) * e_3)$ [ZiCh90].

- **Détection et substitution de variable inductive**

  Une variable inductive est une variable scalaire dont la valeur est fonction

des indices de boucles. Elle est utilisée souvent pour simplifier les expressions des indices de tableau. Par exemple,

```
S = 0
DO I = 1, N
   S = S + 3
   A(S) = A(S-1) + 1
ENDDO
```

Dans ce programme, `S` est une variable inductive. Son expression en fonction de l'indice de boucle `I` est `3*I`. Pour simplifier le test de dépendances, on substitue `S` présent dans l'indice du tableau A par son expression. Le code devient :

```
S = 0
DO I = 1, N
   A(3*I) = A(3*I-1) + 1
ENDDO
S = 3*N
```

– **Propagation de constante**

Une constante symbolique est une variable scalaire dont la valeur est connue à la compilation et n'est pas modifiée pendant l'exécution du programme. L'optimisation suivante donne un exemple de propagation de constante.

```
N = 10                              N = 10
M = 5                               M = 5
DO I = 1, N              ⟹         DO I = 1, 10
  A(I) = A(M) + I                     A(I) = A(5) + I
ENDDO                               ENDDO
```

– **Détection de réduction**

Une réduction est une fonction qui calcule une valeur scalaire sur un vecteur. Certaines dépendances cycliques sur une instruction correspondent à des réductions, ex. somme, produit, calcul du maximum ou minimum d'une série de vecteurs. Ces opérations peuvent être extraites à l'extérieur des boucles et être exprimées par des instructions de réduction. Prenons un

exemple,

```
S = 0
DO I = 1, N
   S = S + A(I)
ENDDO
```

Cette boucle est équivalente à la réduction `SUM : S = SUM (A (1:N))`.

Transformations de boucles

Les opérations de transformation de boucles permettent de reconstruire un corps de boucles équivalent où l'ordre des itérations ou même des instances d'instructions peut avoir été modifié. Le but de ces transformations est d'améliorer l'exploitation et l'explicitation du parallélisme implicite dans la boucle. Une transformation est *légale* si l'exécution du programme transformé donne le même résultat que le programme original (i.e. conserve toutes les dépendances). Une transformation est *valable* pour une boucle si elle est légale et apporte du parallélisme.

De nombreuses transformations ont été proposées dans la littérature. Nous exposons certaines d'entre elles : la distribution de boucle, la fusion de boucle, l'échange de boucles, l'inversion de boucle *loop reversal*, le décalage de boucle et le *strip-mining*.

- La **distribution de boucle** distribue les instructions de la boucle par blocs de **cfc** [3] dans le graphe de dépendances. Pour être légale, aucune dépendance cyclique entre blocs ne doit exister. Le code est transformé en une séquence de nids de boucles dont seuls les corps sont différents. Cette transformation est surtout utilisée pour permettre la vectorisation [AlKe87] et la parallélisation partielle des instructions du corps de boucles. Nous donnons un exemple :

3. **composantes fortement connexes**

```
        DO I = 1, N
S1          T1(I) = 0
S2          T2(I) = T2(I) + T1(I+1)
        ENDDO
```

Cette boucle n'est pas parallèle, parce qu'il existe une anti-dépendance de profondeur 1 de `S2` vers `S1`.

Après distribution, on obtient une séquence de deux boucles parallèles :

```
        DOALL I = 1, N
S2          T2(I) = T2(I) + T1(I+1)
        ENDDOALL

        DOALL I = 1, N
S1          T1(I) = 0
        ENDDOALL
```

– La **fusion de boucle** est la transformation inverse de la distribution de boucle [Wolf89].

Deux nids de boucle adjacents `l1, l2` dont les boucles sont identiques peuvent être fusionnés en un seul nid de boucle, s'il n'existe pas de relation de dépendance d'une instruction `S1` de la boucle `l1` vers une instruction `S2` de `l2` de direction "$>$" [Wolf89].

Cette transformation ne permet pas d'obtenir plus de parallélisme. Elle permet par contre de diminuer l'*overhead* de contrôle et d'augmenter la localité des données accédées. Ces deux derniers effets sont des effets négatifs de la transformation précédente.

– L'**échange de boucles** modifie l'ordre de parcours de l'espace des itérations en échangeant l'ordre d'exécution de deux boucles. Cette transformation est largement utilisée pour améliorer la vectorisation (en déplaçant la dépendance vers la boucle externe) et augmenter la taille des tâches parallèles (en déplaçant la boucle parallèle à l'extérieur) [Wolf89].

Voici un exemple simple :

```
DO I = 2, N
    DO J = 1, M
        A(I,J) = A(I-1,J-1)+A(I,J-1)
    ENDDO
ENDDO
```

Dans ce programme, il y a deux dépendances avec des *ddv* (<,<) et (=,<). Aucune boucle n'est parallèle. Après échange des boucles I et J, on obtient un programme dont la boucle interne est parallèle et vectorisable.

```
DO J = 1, M
    DOALL I = 2, N
        A(I,J) = A(I-1,J-1)+A(I,J-1)
    ENDDO
ENDDOALL
```

La condition de validité de cette transformation est le maintien de la lexico-positivité des vecteurs de dépendance. Elle ne dépend que des informations contenues dans les ddvs.

– Le **décalage de boucle (*loop skewing*)** applique une translation aux itérations de la boucle correspondant à une réindexation des itérations. L'association de cette transformation qui en elle-même ne change rien à l'ordre des itérations avec l'*échange de boucles* est équivalente à une méthode de parallélisation, *la méthode hyperplane* [Lamp74]. Elle peut aussi être associée à d'autres transformations telles que: *strip mining* et l'inversion de boucle, pour exploiter le parallélisme dans les boucles imbriquées.

  Prenons un exemple:

```
DO I = 2, N
    DO J = 2, M
        A(I,J) = A(I-1,J) + A(I,J-1)
    ENDDO
ENDDO
```

Il existe deux dépendances dont les vecteurs de distance sont (0,1) et (1,0). Ces deux dépendances empêchent la parallélisation des deux boucles. Si on décale la boucle J par rapport à la boucle I avec un facteur 1 (i.e. $J' = J+I$),

nous obtenons le nouveau programme:

```
DO I = 2, N
    DOALL J = I+2, I+M
        A(I,J-I) = A(I-1,J-I) + A(I,J-I-1)
    ENDDO
ENDDO
```

Après échange des deux boucles, la boucle I peut être parallélisée. On obtient alors le programme suivant:

```
DO J = 4, N+M
    DOALL I = MAX(2,J-M), MIN(N,J-2)
        A(I,J-I) = A(I-1,J-I) + A(I,J-I-1)
    ENDDO
ENDDO
```

– L'**inversion de boucle** (***loop reversal***) inverse l'ordre des itérations d'une boucle. Cette transformation est légale si elle ne porte aucune dépendance autre qu'une relation de type réduction. Elle est usuellement appliquée avec d'autres transformations (ex. l'échange de boucles) afin d'effectuer une transformation complexe sur un programme.

Prenons un exemple:

```
DO I = 2, N
   DO J = 2, N
      DO K = 1, N-1
         A(I,J,K) = A(I-1,J,K+1) + A(I,J-1,K+1)
      ENDDO
   ENDDO
ENDDO
```

Il existe deux dépendances dont les vecteurs de distance sont (1,0,-1) et (0,1,-1). Ces deux dépendances empêchent la parallélisation des boucles I et J. Après inversion de la boucle K et le déplacement de cette boucle à l'extérieur (qui correspond à deux échanges de boucles : échange des boucles I et K puis des boucles J et I), les vecteurs de distance deviennent (1,1,0) et (1,0,1). Maintenant on peut paralléliser les deux boucles internes, (les

boucles I et J).

```
DO K = N-1, 1, -1
   DOALL I = 2, N
     DOALL J = 2, N
        A(I,J,K) = A(I-1,J,K+1) + A(I,J-1,K+1)
      ENDDOALL
   ENDDOALL
ENDDO
```

– Le *strip-mining* partitionne les itérations d'une boucle en blocs. Cette transformation est toujours légale puisque elle ne change pas l'ordre initial d'ordonnancement des itérations. L'utilisation de cette transformation permet de diviser `n` itérations d'une boucle à $\lceil n/b \rceil$ nombre de blocs (*chunks*) de la même taille (`b itérations`). La taille de bloc est choisi selon la longueur maximale de vecteur d'une machine vectorielle ou le nombre des processeurs disponibles d'un multiprocesseur. Cette transformation permet aussi de mieux exploiter le parallélisme. Par exemple, si la distance minimale de dépendances dans le corps de boucles est supérieure à `n`, la boucle interne peut être parallélisée [Wolf89].

1.3.4 Vectorisation

Dans cette partie, nous présentons le principe de vectorisation. Le but de la vectorisation est de transformer un maximum d'instructions vectorisables en instructions vectorielles tout en respectant toutes les dépendances.

On partitionne d'abord le graphe de dépendances en **composantes fortement connexes** (**cfc**) ou π-blocs selon la terminologie de Kuck [Kuck77] par l'algorithme de Tarjan [Tarj72]. Le nid de boucles initial peut être distribué en autant de petites boucles, par **distribution de boucle**, que le nombre de cfcs. Etant donné que toute instruction simple peut être vectorisée si elle ne dépend pas d'elle-même, la condition nécessaire pour qu'une boucle soit vectorisable est que le graphe de dépendances ne contienne pas de cycle. Dans ce cas, toutes les instructions contenues dans la boucle peuvent être vectorisées après distribution de la boucle.

Lorsqu'il y a un cycle de dépendances, les boucles englobant cette cfc cyclique doivent normalement rester séquentielles. Plusieurs transformations ont été proposées dans le but de casser un cycle de dépendances afin d'en extraire des instructions vectorielles. Par exemple, l'expansion de variables scalaires, ou bien *index set splitting* [Wolf89] qui consiste à diviser les itérations d'une boucle en deux parties indépendantes.

Si un cycle de dépendances se trouve dans un corps de boucles imbriquées, la stratégie de vectorisation partielle proposée par Allen & Kennedy [AlKe87] peut être appliquée. Elle permet de vectoriser partiellement les instructions englobées dans le cycle. L'idée essentielle de cette méthode est de conserver "séquentielle" la boucle dont le graphe de dépendances possède un cycle et de paralléliser les autres boucles. Cette méthode de vectorisation doit s'effectuer des boucles extérieures vers les boucles les plus internes de la même manière. Les graphes réduits successifs ne comportent plus les dépendances portées par les boucles précédentes (i.e. toutes les dépendances de profondeur inférieure sont supprimées).

Nous illustrons ci-après la vectorisation pour l'exemple 1.3. Observons le graphe de dépendances[4] (figure 1.6), il contient deux π-blocs : un cycle composé par `S1` et `S2`, une instruction isolée `S3`.

Dans un premier temps, après distribution, `S3` est vectorisée. Les boucles englobant `S1` et `S2`, étant dépendantes l'une de l'autre, restent séquentielles.

```
        DO I = 1, N
           DO J = 1, M
S1:            A(I+1,J) = A(I+1,J) + B(I,J-1)
S2:            B(I,J) = B(I,J) + A(I,J)
           ENDDO
        ENDDO
S3:     C(1:N,1:M) = A(2:N+1,0:M-1) + B(0:N-1,1:M)
```

Considérons maintenant les boucles non vectorisées. Puisque le graphe de dépendances sur la boucle `I` est un cycle, la boucle `I` doit rester séquentielle. Après avoir supprimé les dépendances de profondeur 1, le graphe de dépendances réduit pour la boucle `J` est illustré par la figure 1.7.

4. un π-bloc est représenté par une *boîte en pointillés*

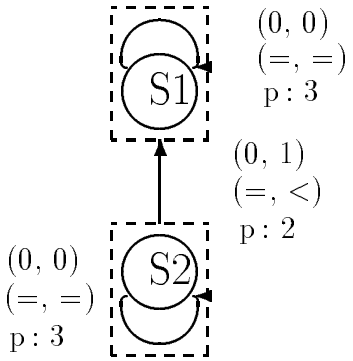


FIG. 1.7 - *Le graphe de dépendances réduit de l'exemple 1.3*

La boucle `J` peut être vectorisée puisqu'il n'y a plus de cycle dans le graphe de dépendances.

```
        DO I = 1, N
S2:             B(I,1:M) = B(I,1:M) + A(I,1:M)
S1:             A(I+1,1:M) = A(I+1,1:M) + B(I,1:M)
        ENDDO
S3:     C(1:N,1:M) = A(2:N+1,0:N-1) + B(1:N,1:M)
```

L'échange de boucles peut améliorer la vectorisation dans le cas où les boucles les plus internes ne peuvent pas être vectorisées. Prenons l'exemple d'un nid de boucles imbriquées `I` et `J` :

```
DO I = 1, N
   DO J = 1, M
       A(I,J+1) = A(I,J)*B(I,J) + C(I,J)
   ENDDO
ENDDO
```

Il y a une dépendance de vecteur de dépendance constant (0,1) dans le corps de boucles.

Pour pouvoir utiliser les instructions vectorielles, les boucles internes doivent être vectorisées, dans tous les cas où cela est possible. Si on échange les boucles `I` et `J`, le *ddv* des dépendances devient (<,=). La boucle vectorisable `I` est maintenant interne. Le code devient :

```
DO J = 1, M
   A(1:N,J+1) = A(1:N,J)*B(1:N,J) + C(1:N,J)
ENDDO
```

Cette transformation permet aussi d'améliorer la localité spatiale.

1.3.5 Parallélisation

Après avoir étudié les techniques de vectorisation, nous présentons, dans cette section, le principe de parallélisation des boucles.

De même que le but de la vectorisation est de détecter les instructions vectorisables dans les boucles afin de les transformer sous forme vectorielle, le but de la parallélisation est de détecter les boucles parallélisables afin de les transformer en instructions parallèles.

Deux formes de boucle parallèle ont été proposées : `DOALL` et `DOACROSS` [Cytr84]. Dans une boucle `DOALL` toutes les itérations de la boucle peuvent être exécutées parallèlement sur multi-processeurs sans synchronisation. L'exécution des différentes itérations d'un boucle `DOACROSS` avec un délai `d` s'effectue partiellement en parallèle : il existe un délai `d` entre l'exécution d'une itération et la suivante. Nous nous intéressons ici particulièrement à la parallélisation sous forme de `DOALL`, c'est à dire la parallélisation sans synchronisation.

Une boucle est parallèle si elle ne *porte* pas de dépendance.

Prenons un exemple :

```
DO I = 1, N
   DO J = 1, M
      T1(I,J) = T2(I,J)
      T2(I,J) = T2(I,J) + T1(I,J-1)
   ENDDO
ENDDO
```

FIG. 1.8 - *Exemple 1.4*

Il y a deux dépendances dans ce nid de boucles caractérisées par les *ddvs* (=,=) et (=,<). Aucune dépendance n'est portée par la boucle `I` (de profondeur 1). La parallélisation naturelle sans transformation de programme préalable donne le résultat suivant :

```
DOALL I = 1, N
   DO J = 1, M
      T1(I,J) = T2(I,J)
      T2(I,J) = T2(I,J) + T1(I,J-1)
   ENDDO
ENDDOALL
```

De nombreuses transformations ont été proposées pour améliorer l'exploitation du parallélisme , entre autres : distribution de boucles, échange de boucles, *loop skewing*, inversement de boucle, blocage de boucles ... [Wolf89]. Ces différentes techniques ont pour but d'exploiter les différents types de parallélisme et de tenir compte des caracteristiques du code et de la machine cible. La parallélisation d'un programme est composée d'une série de transformations qui permet l'obtention d'un code parallèle optimal. Sachant que si on effectue `n` transformations sur un programme en utilisant un ordre différent à chaque fois, on obtient n! programmes équivalents en sémantique mais différents en syntaxe et possédant des quantités de parallélisme différentes, l'ordre d'application des transformations est très important. Toutefois trouver l'ordre optimal dans lequel les transformations doivent être appliquées est difficile.

Récemment, le concept de transformation unimodulaire a été introduit. Les transformations de réordonnancement d'un espace I^n dans I^n peuvent être caractérisées par une matrice unimodulaire. Certaines transformations sont des transformations unimodulaires élémentaires telle que : l'échange de boucles, *loop skewing* et l'inversion de boucle. La combinaison de transformations unimodulaires est encore unimodulaire et correspond au produit des matrices des transformations. L'avantage de ce concept est qu'il permet de caractériser par une seule matrice de transformation l'ensemble des transformations.

Nous présentons maintenant certaines stratégies de parallélisation classiques telles que la méthode étendue d'Allen & Kennedy, la méthode hyperplane et les méthodes de partitionnement.

Méthode étendue d'Allen & Kennedy

La méthode de vectorisation proposée par Allen & Kennedy [AlKe87] peut être employée également pour la parallélisation. Les boucles naturellement parallèles (i.e. pour lesquelles les composantes des ddvs sont égales à "=") sont détectées et déplacées à l'extérieur. L'algorithme d'Allen & Kennedy est ensuite appliqué pour paralléliser les autres boucles. Il procède boucle par boucle en commençant des boucles les plus externes vers les boucles les plus internes. La parallélisation de chaque boucle est effectuée à partir du graphe de dépendances réduit dans lequel les dépendances portées par les boucles précédentes sont supprimées. Le graphe de dépendances est distribué en composantes fortement connexes. Les parties acycliques sont parallélisées. Pour chaque cycle, on conserve "séquentielle" la boucle portant la dépendance et on parallélise les boucles internes, et ainsi de suite. Cette technique de parallélisation est basée sur la transformation de distribution des boucles.

Reprenons l'exemple 1.4, après avoir appliqué la méthode d'Allen & Kennedy, on obtient le code parallèle :

```
DOALL I = 1, N
   DOALL J = 1, M
      T1(I,J) = T2(I,J)
   ENDDOALL
   DOALL J = 1, M
      T2(I,J) = T2(I,J) + T1(I,J-1)
   ENDDOALL
ENDDOALL
```

Ce programme contient plus de parallélisme que dans la version sans transformation.

Cette méthode est implantée dans la plupart des paralléliseurs, entre autres, **PARAFRASE2** [PoGi89], **PTRAN** [ABCC87], **PIPS** [IJT91], **PAF** [TDF87].

Méthode hyperplane

La méthode hyperplane [Lamp74] [IrTr88a] [IrTr88b] est une méthode de parallélisation pour les boucles imbriquées. Elle génère du code parallèle où la première boucle est séquentielle et les autres sont parallèles. Elle correspond à une transformation unimodulaire représentant une combinaison d'une suite de transformations unimodulaires élémentaires : la permutation de boucles (l'échange de boucles est un cas particulier), l'inversion de l'ordre d'exécution d'une boucle, et le décalage de boucle. L'idée de base est de concentrer toutes les dépendances sur la première boucle et de conserver cette boucle séquentielle, afin que les autres boucles puissent être exécutées en parallèle. Du point de vue mathématique, la méthode hyperplane essaie de trouver une série d'hyperplans tels que toutes les itérations d'un hyperplan puissent être exécutées en parallèle.

```
      DO I = 1, N
        DO J = 1, M
S:          T(I,J) = T(I-1,J) + T(I,J-1)
        ENDDO
      ENDDO
```

FIG. 1.9 - *Exemple 1.5*

Dans l'exemple 1.5, il existe deux dépendances cycliques sur l'instruction S.

Les *ddvs* sont (<,=) et (=,<). La conservation de la boucle I séquentielle ne permet pas d'ôter toutes les dépendances car le graphe de dépendances réduit sur la boucle J possède encore un cycle. La méthode d'Allen & Kennedy ne permet pas la parallélisation de l'une des boucles I ou J.

Or après la transformation unimodulaire : $I' = I + J, J' = J$, on obtient

```
DO I' = 2, N + M
  DO J' = max(1, I' − N), min(M, I' − 1)
     T(I' − J', J') = T(I' − J' − 1, J') + T(I' − J', J' − 1)
  ENDDO
ENDDO
```

Les dépendances deviennent (<,=), (<, <). Il est clair que la boucle J' est maintenant parallèle alors que la boucle I' doit rester séquentielle.

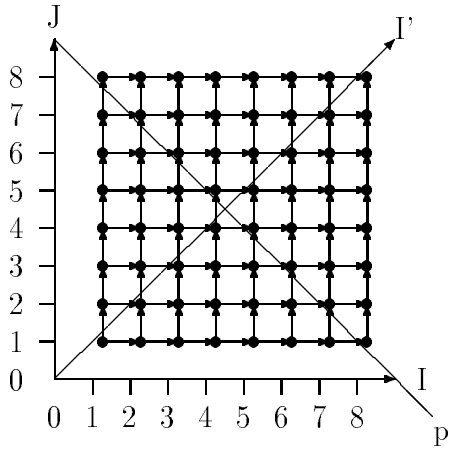


FIG. 1.10 - *L'espace d'itérations de l'exemple 1.5*

La figure 1.10 illustre l'espace d'itération de l'exemple 1.5. On suppose que `N` et `M` valent 8. Dans cette figure, I' donne la direction d'ordonnancement. Toutes les itérations appartenant à un même hyperplan (orthogonal à I') peuvent être exécutées en parallèle. Par exemple, le plan `p` est constitué de 8 itérations parallèles.

L'utilisation de la méthode hyperplane permet parfois la parallélisation de boucles qui ne sont pas parallélisées par la methode d'Allen & Kennedy. Cependant, cette méthode ne s'applique pas toujours. Notament, pour les cas où il n'y a qu'une seule boucle ou quand l'envelope convexe de toutes les distances de dépendance n'est pas lexico-positive (voir la section 5).

Méthodes de Partitionnement

Ce type de méthode est basé sur une combinaison de transformations de réordonnancement non-unimodulaires (i.e. une transformation d'un espace d'itérations de dimension `n` dans un espace de dimension `p` avec `p` > `n`). L'idée essentielle est de partitionner l'espace d'itérations en blocs identiques tels que les dépendances soient toutes respectées et regroupées (1) soit au niveau du bloc de manière à conserver le parallélisme à l'intérieur du bloc, (2) soit au niveau des itérations pour que les blocs eux-mêmes soient parallélisables. Ce type de méthode est particulièrement intéressant pour diminuer l'*overhead* de contrôle et augmenter la localité des données accédées.

De nombreuses méthodes ont été proposées dans la littérature, entre autres : la méthode *cycle shrinking* [Poly88] (basée sur les transformations *strip-mining* et l'échange de boucles, et utilisant la distance minimale comme information de dépendance), la méthode *supernœud* de F. Irigoin (basée sur la méthode hyperplane et utilisant le cône de dépendances) [IrTr87], et la méthode de Peir et Cytron [PeCy87] (basée sur l'ordonnancement de récurrence linéaire et utilisant les vecteurs de dépendance). Pour des raisons de simplicité, nous prenons dans la suite de cette section la méthode *cycle shrinking* comme modèle.

La méthode *cycle shrinking* partitionne l'espace d'itération en blocs de forme rectangulaire. Le schéma de transformation pour une boucle simple est illustré par la figure suivante. Une boucle est divisée en deux boucles.

```
DO I = 1, N  ⟹  DO IB = 1, N, K
                    DO I = IB, MIN (N, IB+K-1)
```

Après transformation, on dit que la boucle `IB` est la "boucle du bloc", et la boucle `I` est la "boucle d'éléments". Si on choisit la taille du bloc proprement de telle

sorte que toutes les dépendances soient *portées* par la "boucle du bloc", on obtient une boucle d'éléments parallèle. On choisit pour la valeur K la valeur minimum des distances des dépendances portées par cette boucle. K doit être supérieur à 1 pour que cette méthode soit réellement rentable. Le parallélisme est fonction de K. Les principes de cette méthode sont détaillés dans [Poly88].

Prenons un exemple simple :

```
          DO I = 1, N
S1:          A(I+4) = B(I) + C(I)
S2:          B(I+3) = A(I) * C(I)
          ENDDO
```

Il y a deux dépendances cycliques uniformes ($d1 = 3, d2 = 4$). La méthode d'Allen & Kenndy ne permet pas de paralléliser cette boucle puisqu'il existe un cycle de dépendances. La méthode hyperplane ne peut pas être utilisée non plus car elle ne fonctionne que pour des boucles parfaitement imbriquées. Toutefois si on partitionne la boucle en prenant K = 3, la "boucle d'éléments" devient parallèle.

```
          DO IB = 1, N, 3
             DOALL I = IB, MIN(N, IB+2)
S1:             A(I+4) = B(I) + C(I)
S2:             B(I+3) = A(I) * C(I)
             ENDDOALL
          ENDDO
```

Méthode d'ordonnancement linéaire

La parallélisation d'une boucle peut être vue aussi comme un problème d'ordonnancement des itérations ou des instructions de la boucle. Un réordonnancement linéaire *monodimensionnel* est une projection (exprimée par une fonction de temps linéaire) de l'ensemble des itérations/instructions multidimensionnelles sur un espace de temps unidimensionnel. Et un réordonnancement linéaire *multidimensionnel* est une projection (exprimée par un vecteur de fonctions de temps linéaire) de l'ensemble des itérations/instructions multidimensionnelles sur un

espace de temps multidimensionnel. La méthode hyperplane est un réordonnancement linéaire monodimensionnel des itérations de boucle.

De nombreux algorithmes ont été proposés dans la litérature, entre autres : F.Irigoin a proposé le calcul d'un ordonnancement linéaire monodimensionnel à partir des informations contenues dans le cône de dépendances d'une boucle [IrTr88b]; W.Shang & J.Fortes ont proposé le calcul d'un ordonnancement linéaire monodimensionnel permettant l'obtention d'un *temps* optimal en présence de dépendances uniformes [ShFo91]; la méthode de *affine-by-statement scheduling* proposée par A.Darte & Y.Robert [DaRo92] [DaRR92] utilise différents ordonnancements affines pour les instructions; l'algorithme de P.Feautrier [Feau92] permet de calculer des ordonnancements affines ou quasi-affines à un niveau plus fin : les instances des instructions.

Ce type de méthode permet de conduire généralement à un résultat optimal. Leur complexité théorique est, toutefois, généralement exponentielle.

Comparaison des méthodes de parallélisation

Aucune méthode n'est universelle et ne marche parfaitement pour tous les cas. Chaque méthode possède son cadre d'application pour lequel elle est plus rentable que toutes les autres. Le choix d'une méthode de parallélisation optimale d'un programme reste un problème ouvert. Il dépend de la complexité de son graphe de dépendances. Informellement, s'il y a pas de dépendance, le corps de boucles est naturellement parallèle; s'il y a pas de cycle résidant à tous les niveaux de boucles, la méthode d'Allen & Kennedy permet d'exploiter le parallélisme; si le cycle contient les dépendances de type *anti-dependence*, la transformation *node splitting* permet parfois de le casser; si ce cycle ne peut pas être brisé, une des méthodes de partitionnement peut être essayée, et dans le cas des boucles imbriquées la méthode hyperplane peut éventuellement être utilisée.

Les méthodes de parallélisation basées sur le réordonnancement linéaire permettent généralement l'obtention d'un fort degré de parallélisme. Cependant, leur calcul conduit à un problème de programmation linéaire en nombres entiers et leur complexité théorique est exponentielle.

1.4 Conclusion

La plupart des paralléliseurs se concentrent sur les techniques d'optimisation et de reconstruction du programme. Ils détectent le parallélisme potentiel et le traduisent à l'aide de primitives dans le programme. Pour utiliser au mieux le parallélisme détecté, le programme parallèle doit tenir compte des caractéristiques des machines cibles. Des problèmes de gestion mémoire pour les multiprocesseurs à mémoire hiérarchisée [Anco91], de réordonnancement des boucles et d'allocation des processeurs [SMC91] [TaFa92], de synchronisation et de communication interviennent [Li91] aussi. Le détail des études [Poly88] dont ils ont fait l'objet dépasse le cadre de cette thèse.

L'étude de la vectorisation et la parallélisation automatique a débutée vers 1970. De nombreux compilateurs/vectoriseurs commerciaux existent maintenant pour les machines vectorielles. Des résultats statistiques indiquent que, pour la plupart, ils vectorisent automatiquement 77% des boucles vectorisables [LCD91]. Certains systèmes de parallélisation automatique sont aussi commercialisés comme : *fpp* [Cray90], *Forge* [Forg90], *KAP* [Kuck89] et *VAST* [EiBl91]. Cependant, leur taux de parallélisation automatique n'est pas le même. Les statistiques indiquent que les techniques d'analyse des programmes et de transformations actuelles ne sont pas encore suffisamment avancées pour l'obtention de bonne performance [ChPa91] [EiBl91] [PePa92a]. De nombreuses recherches dans ce domaine doivent donc encore être effectuées. Elles sont entreprises par de grandes universités et centres de recherches : **PARAFRASE-2** à *University of Illinois* [PoGi89]; **PARASCOPE, PTOOL** à *Rice University* [KMT91]; **PTRAN** à *IBM Research Center* [ABCC87]; **SUIF** à *Stanford University* [TWLPH91]; **TINY** à *Oregon Graduate Institute* [Wolf91b]; **PAF** à l'université de Paris 6 [TDF87]; **PIPS** à l'Ecole des mines [IJT91].

Rappelons que la parallélisation débute généralement par le calcul du graphe des dépendances. De nombreux algorithmes ont été proposés pour les calculer. Les transformations de programme, pouvant modifier la sémantique du programme, imposent des abstractions des dépendances spécifiques. Plusieurs ont aussi été proposées : *profondeur, ddv, dc, distance*. Quelles sont les différences entre les

différents algorithmes de calcul des dépendances? Quel est le comportement dans la pratique d'un algorithme approximatif dont la complexité théorique est exponentielle (Fourrier-Motzkin)? Quels sont les liens entre les transformations et les abstractions des dépendances? Ce sont ces questions auxquelles nous avons tenté de répondre dans cette thèse.

Chapitre 2

Test de dépendance

Le test de dépendance est une des phases les plus importantes pour la détection et l'exploitation du parallélisme implicite des programmes. Sa précision et son efficacité conditionnent directement le succès de la phase de parallélisation. Le problème principal du test de dépendance est de tester si deux instructions référencent un même emplacement mémoire. C'est un problème indécidable dans le cas général à la compilation. Une référence indirecte à un tableau du type `A(B(I))` fait partie des exemples où l'on ne connait pas à la compilation l'ensemble des éléments qui seront accédés lors de l'exécution. Il peut conduire aussi à la résolution de systèmes très *complexes* tel que : rechercher si l'équation $a^n = b^n + c^n$ [Mayd92] admet une solution. Les algorithmes permettant de traiter des systèmes particuliers étant relativement coûteux, cette classe de problèmes ne rentre pas dans l'ensemble de cas traités par les tests de dépendances classiques à savoir les cas linéaires. Sous l'hypothèse que toutes les fonctions d'accès aux éléments des tableaux ainsi que les bornes de boucles sont linéaires, le test de dépendance a pour but de rechercher une solution entière au problème exprimé sous la forme d'un système linéaire et de caractériser de mannière finie ces solutions. De nombreux algorithmes, exacts ou approximatifs, de complexité polynômiale ou exponentielle, ont été proposés dans la littérature. Nous présentons ces différents algorithmes ainsi que les compromis *précision/complexité* effectués par les différents vectoriseurs/paralléliseurs. Nous verrons l'importance d'une étude expérimentale pour évaluer la complexité *pratique* et pour optimiser la précision d'un algorithme.

Dans ce chapitre, nous détaillons successivement : le concept de dépendance, le problème du test de dépendance et les différentes méthodes de l'analyse des dépendances. Nous présentons, dans la section 2.4 le système d'analyse des dépendances du paralléliseur **PIPS** qui a été développé au CRI à l'Ecole des Mines de Paris à Fontainebleau depuis 1988 [IJT91] et que nous avons amélioré et instrumenté. Nous présentons enfin, dans la section 2.5, les résultats de l'évaluation de performance de ce test avant de conclure.

Les différentes abstractions des dépendances caractérisant les ensembles d'itérations *dépendantes* sont présentées dans le chapitre 3.

2.1 Dépendance

La sémantique d'un programme écrit en language 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[1]. Deux instructions peuvent être exécutées en parallèle si ces deux instructions n'accèdent pas le même emplacement mémoire[2]. Dans ce cas, on dit que les deux 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, nous présentons le concept de dépendance et en donnons une définition, tout particulièrement pour le cas des nids de boucles (car elles contiennent l'essentiel du parallélisme implicite). Ensuite nous détaillons le problème du test de dépendance et en donnons une formulation mathématique.

1. en l'absence d'*aliasing*
2. nous n'étudions que les dépendances de données dans cette thèse

2.1.1 Concept de dépendance de donnée

Nous décrivons, dans cette section, les trois types de dépendance définis à l'origine par Kuck [Kuck78].

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

- `s et t` référencent `M` et l'une au moins de ces deux instances le modifie;
- selon l'ordre d'exécution séquentiel `s` est exécutée avant `t`.

Nous distinguons trois types de dépendance, conditionnant l'exécution paralléle d'un programme.

La dépendance de `S` vers `T` est

- une *flow-dependence* si `s` modifie `M` et `t` lit `M` . Elle est notée $S\delta^f T$;
- une *anti-dependence* si `s` lit `M` et `t` modifie `M`. Elle est notée $S\delta^a T$;
- une *output-dependence* si `s` et `t` modifient toutes deux `M`. Elle est notée $S\delta^o T$.

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

Nous considérons maintenant le cas des nids de boucles `DO`. Le schéma général d'un nid de boucles est représenté en figure 2.1. Il illustre le modèle suivant (sous les hypothèses de linéarité des fonctions) :

- les boucles i_1, ...i_e,i_{e+1} ... i_m englobent une instruction S (notée $S(I, I_s)$), les boucles i_1 ...i_e, i_{m+1} ...i_n englobent l'instruction T (notée $T(I, I_t)$).
- S, T accèdent au même tableau A.
- le tableau A est déclaré : $A(L_1{:}U_1, L_2{:}U_2, ..., L_d{:}U_d)$.
- f_1, ... , f_d, g_1, ... , g_d sont les fonctions d'indices du tableau qui sont des fonctions linéaires de variables scalaires entières.

$$
\begin{array}{l}
do\ \ i_1 \ = \ l_1\ ,\ u_1 \\
do\ \ i_2 \ = \ l_2\ (i_1)\ ,\ u_2\ (i_1) \\
. \\
. \\
. \\
do\ \ i_e \ = \ l_e\ (i_1, ..., i_{e-1})\ ,\ u_e\ (i_1, ..., i_{e-1}) \\
\quad do\ \ i_{e+1} \ = \ l_{e+1}\ (i_1, ..., i_e)\ ,\ u_{e+1}\ (i_1, ..., i_e) \\
\quad . \\
\quad . \\
\quad . \\
\quad do\ \ i_m \ = \ l_m\ (i_1, ..., i_{m-1})\ ,\ u_m\ (i_1, ..., i_{m-1}) \\
\quad S\ (i_1, ..., i_e, i_{e+1}, ..., i_m)\ :\ A\ (f_1\ (i_1, ..., i_m), f_2\ (...), ..., f_d\ (...))\ =\ ... \\
\quad enddo\ \{e+1, ..., m\} \\
\quad . \\
\quad . \\
\quad . \\
\quad do\ \ i_{m+1} \ = \ l_{m+1}\ (i_1, ..., i_e)\ ,\ u_{m+1}\ (i_1, ..., i_e) \\
\quad . \\
\quad . \\
\quad . \\
\quad do\ \ i_n \ = \ l_n\ (i_1, ..., i_e, i_{m+1}, ..., i_{n-1})\ ,\ u_n\ (i_1, ..., i_e, i_{m+1}, ..., i_{n-1}) \\
\quad T(i_1, ..., i_e, i_{m+1}, ..., i_n)\ :\ ...\ =\ A\ (g_1\ (i_1, ..., i_e, i_{m+1}, ..., i_n), g_2\ (...), ..., g_d\ (...)) \\
\quad enddo\ \{m+1, ..., n\} \\
enddo\ \{1, ..., e\}
\end{array}
$$

FIG. 2.1 - *Schéma d'un nid de boucles*

– l_1, u_1, ..., l_n, u_n sont les bornes inférieures et supérieures des boucles. Elles sont aussi des fonctions linéaires.

Une dépendance entre deux instructions d'un nid de boucles imbriquées est représentée en figure 2.1. Elle est définie de la manière suivante :

Définition 2.1 *L'instruction* `T` ***dépend de*** *l'instruction* `S` *(notée* $S\delta T$*) s'il existe une instance* $S(\vec{i}, \vec{i_s})$ *de* `S`*, une instance* $T(\vec{i'}, \vec{i_t})$ *de* `T`*, et un emplacement mémoire* `M`*:* $A(c_1, ..., c_d)$ *tel que :*

1. $S(\vec{i}, \vec{i_s})$ *et* $T(\vec{i'}, \vec{i_t})$ *accèdent au même emplacement mémoire* `M`*;*

2. *les deux itérations* $(\vec{i}, \vec{i_s})$, $(\vec{i'}, \vec{i_t})$ *appartiennent au domaine d'itérations des boucles;*

3. *selon l'ordre d'exécution séquentiel,* $S(\vec{i}, \vec{i_s})$ *est exécutée avant* $T(\vec{i'}, \vec{i_t})$.

2.1.2 Problème de dépendance

Nous présentons, dans cette section, une formulation des dépendances inspirée de celle de M. Wolfe [Wolf89] et de celle de H. Zima [ZiCh90].

Il ne peut y avoir une dépendance entre deux instructions que si ces deux instructions référencent la même variable. Si le test de dépendance sur les variables scalaires est
relativement simple, il n'en est pas de même pour les références aux éléments d'un tableau. Nous nous concentrons donc, dans cette section, sur l'étude du test de dépendance entre instructions référençant des éléments d'un tableau au sein d'un nid de boucles.

Pour tester l'existence d'une dépendance de S vers T, il faut calculer l'intersection de l'ensemble des éléments référencés par S avec celui des éléments accédés par T.

La formulation mathématique du test de dépendance, d'après la définition 2.1, correspond à la vérification de l'existence d'une solution entière d'un système diophantien (i.e. système linéaire en nombres entiers). Le système est constitué de trois parties (figure 2.2):

1. un ensemble d'équations (2.1) déduites de la condition 1 de la définition 2.1;

2. un ensemble d'inégalités (2.2) déduites de la condition 2 de la définition 2.1;

3. un ensemble des contraintes (2.3) déduites de la condition 3 de la définition 2.1.

$$f_k\ (\vec{i}, \vec{i_s}) \ = \ g_k\ (\vec{i'}, \vec{i_t}) \ \ (1 \leq k \leq d) \qquad (2.1)$$

$$\begin{cases} l_k \ \leq \ i_k \ \leq \ u_k & (1 \leq k \leq e) \\ l_k \ \leq \ i'_k \ \leq \ u_k & (1 \leq k \leq e) \\ l_{s_k} \ \leq \ i_{s_k} \ \leq \ u_{s_k} & (e+1 \leq s_k \leq m) \\ l_{t_k} \ \leq \ i_{t_k} \ \leq \ u_{t_k} & (m+1 \leq t_k \leq n) \end{cases} \qquad (2.2)$$

$$\begin{cases} \exists c \quad (1 \leq c \leq e) \ / & \\ i_k = i'_k & (1 \leq k \leq c) \\ i_{k+1} < i'_{k+1} & si\ k+1 \leq e \end{cases} \qquad (2.3)$$

FIG. 2.2 - *Les composantes du système de dépendances*

Lorsqu'il existe une solution entière au système représenté par la figure 2.2, une dépendance existe. Dans la section suivante, nous présentons les principales méthodes permettant de résoudre ce système de dépendance exactement ou approximativement.

2.2 Analyse de dépendance

Rappelons que le problème du test de dépendance est un problème de décision. Il consiste à vérifier l'existence d'une solution entière à un système. Sous les hypothèses de linéarité décrites précédemment , c'est un problème de programmation linéaire en nombres entiers (notée PE). Cependant ce type de problème est NP-complet [Schr86]. Il existe des algorithmes approximant l'existence d'une solution entière par celle d'une solution réelle. Ces algorithmes peuvent se traduire en général sous la forme d'un problème de programmation linéaire (notée PL) polynômial. Toutefois, les algorithmes de PE et PL tels que : la méthode du simplexe [Schr86] ou la méthode des coupes de Gomory [Gree71] ne sont pas très appropriés pour traiter le problème du test des dépendances car ils apportent une solution exacte au problème alors que l'existence (ou l'inexistence) d'une solution est suffisante. Différentes méthodes spécifiques à l'analyse de dépendance, donnant des solutions exactes ou approchées, ont donc été étudiés.

Bien que plusieurs tests exacts aient été proposés : *PIP* [Feau88], *FAS*^{3}*T* [BePT90] et *Omega* [Pugh92], de nombreux vectoriseurs/parralléliseurs utilisent un test de dépendance simplifié, soluble par des méthodes simples et efficaces bien que moins précises. Ces méthodes sont basées sur la théorie des équations diophantiennes [Bane88], l'évaluation des valeurs minimales et maximales des bornes d'une fonction linéaire [Bane88] ou encore la faisabilité de système linéaire [Kuhn80] [TIF86].

En général, le système de dépendance est composé d'un ensemble d'équations, et d'un ensemble d'inéquations déduit des bornes des boucles. Afin d'obtenir une approximation, plusieurs possibilités nous sont offertes, on peut chercher soit (1) des solutions entières au système d'équations sans tenir compte des inéquations, (2) des solutions réelles au système avec les inéquations, (3) des solutions dimension par dimension, (4) des solutions à l'équation de linéarisation (la combinaison des équations). Ces méthodes approximatives agrandissent l'ensemble des solutions exactes, elles sont toujours conservatives. C'est l'existence d'une solution dans l'ensemble approximé qui conduit à déclarer qu'il y a dépendance.

Avant d'introduire les différentes manières d'approximer l'ensemble des solutions, nous introduisons quelques notations :

Notations :

- $S_{int}\ (P)$: l'ensemble des solutions entières du système P;
- $S_{reel}\ (P)$: l'ensemble des solutions réelles du système P;
- DS : le système de dépendance complet donné par la figure 2.2;
- DS_i : le sous système de dépendance pour le `i`-ième indice du tableau, $DS = \cup_{i=1}^{d} DS_i$;
- Eq : l'ensemble des équations du système DS ((2.1) dans la Fig 2.2);
- Eq_i : l'équation traduisant un conflit mémoire pour le `i`-ième indice du tableau, $Eq = \cup_{i=1}^{d} Eq_i$.

2.2.1 Analyse indice par indice

Etant donné la simplicité du test de dépendance mono-dimensionnel, une des approximations possibles pour étudier une dépendance multi-dimensionnelle est la vérification de l'existence de dépendance indice par indice. Ceci n'est qu'une approximation car l'ensemble des solutions entières du système DS est égal à l'intersection des solutions entières des systèmes DS_i ($S_{int}(DS) = \cap_{i=1}^{d} S_{int}\ (DS_i)$) et que l'ensemble des solutions entières du système DS_i est inclus dans celui des solutions entières du système d'équation Eq_i, nous avons la relation $S_{int}(DS) \subseteq S_{int}\ (Eq_i)$. Comme l'ensemble des solutions entières du système DS_i est contenu dans celui des solutions réelles, nous avons la seconde relation $S_{int}(DS) \subseteq S_{reel}(DS_i)$.

Au cours d'une analyse indice par indice, le système est déclaré *sans solution* (pas de dépendance) aussitôt que le test de faisabilité en réels sur une dimension est négatif (i.e. $S_{int}(DS_i)$ ou $S_{int}\ (Eq_i)$ ou $S_{reel}(DS_i) = \emptyset$), puisque cela prouve que $S_{int}(DS)$ est *vide*.

Nous présentons maintenant trois tests mono-dimensionels.

Test du Greatest Common Divisor (GCD) [Bane76] :

Le test du GCD permet de savoir si une équation linéaire diophantienne sous la forme (2.4) admet une solution entière. Il est utilisé pour calculer $S_{int}\ (Eq_i)$.

$$a_1x_1 + a_2x_2 + ... + a_nx_n = c \qquad (2.4)$$

L'équation (2.4) admet une solution entière si et seulement si $gcd(a_1, a_2, ..., a_n)$ $divise$ c. C'est un test simple mais pas très sélectif, parce que dans la plupart des cas le $gcd(a_1, a_2, ..., a_n)$ vaut 1. Il ne permet pas, non plus, de conclure à l'inexistence de solution (pas de dépendance) si l'équation (2.4) admet des solutions entières qui sont toutes en dehors des bornes du domaine.

Le coût de ce test étant relativement faible (complexité polynômiale), il est utilisé par la plupart des tests de dépendance des paralléliseurs.

Test de Banerjee [Bane79] [Bane88] :

Le test de Banerjee permet de savoir si une équation linéaire bornée admet une solution *réelle* dans le domaine (2.5) où $l_i,\ u_i (1 \leq i \leq n)$ sont constantes. Il est utilisé pour calculer $S_{reel}(DS_i)$.

$$\begin{cases} a_1x_1 + a_2x_2 + ... + a_nx_n = c \\ l_1 \leq x_1 \leq u_1 \\ l_2 \leq x_2 \leq u_2 \\ ... \\ l_n \leq x_n \leq u_n \end{cases} \qquad (2.5)$$

Ce test est basé sur le theorème des valeurs intermédiaires :
Soit $F(X) = a_1x_1 + a_2x_2 + ... + a_nx_n$; et $min(F(X)),\ max(F(X))$ les minimum et maximum de la fonction $F(X)$ en fonction des bornes de $x_1, ..., x_n$. **Le système (2.5) admet une solution réelle si et seulement si** $min(F(X)) \leq c \leq \max(F(X))$.

Le test de Banerjee conduira à la détection d'une *fausse* dépendance lorsque le système admettra au moins une solution réelle mais pas de solution entière.

Ce test a été étendu pour pouvoir être appliqué dans les cas (1) où les bornes des boucles sont des fonctions linéaires des indices de boucles externes (*le test de Banerjee trapézoidal*) [Bane88] (2) et où les boucles sont normalisées avec une borne inférieure à 0 et une borne supérieure symbolique [PePa91].

Le test de dépendance du paralléliseur **PARAFRASE** développé à l'université de l'Illinois est basé principalement sur ce test.

I test [KKP90] [PKK91] :

Le I test est une combinaison des tests du GCD et de Banerjee, c'est aussi un test approximatif. Il recherche l'existence d'une solution entière, en utilisant le test du GCD, et prend en compte les bornes des variables comme Banerjee. Il est donc utilisé pour calculer $S_{int}(DS_i)$.

Un nouveau concept **Intervalle d'equations** (2.6) a été introduit dans [KKP90]

$$a_1x_1 + a_2x_2 + ... + a_nx_n = [L, U] \qquad (2.6)$$

pour répresenter un ensemble d'équations $a_1x_1 + a_2x_2 + ... + a_nx_n = y$ ($L \leq y \leq U$). Un intervalle d'équations est soluble dans le domaine $(l_1, u_1; l_2, u_2; ...; l_n, u_n)$ si au moins une équation est soluble dans ce domaine. Le I test est basé essentiellement sur deux théorèmes.

1. Elimination des variables de l'intervalle d'équations par la méthode de Banerjee, donnant les bornes d'une fonction linéaire [Bane88] :
   l'intervalle d'équations (2.6) est soluble dans le domaine $(l_1, u_1; l_2, u_2; ...; l_n, u_n)$ (où les bornes sont constantes) si et seulement si

   $$a_1x_1+a_2x_2+...+a_{n-1}x_{n-1} = [L-a_n^+u_n+a_n^-l_n,\ U-a_n^+l_n+a_n^-u_n] \qquad (2.7)$$

   est soluble dans le domaine $(l_1, u_1; l_2, u_2; ...; l_{n-1}, u_{n-1})$ [3].

2. Test du GCD pour l'intervalle d'équations :
   Soit $d = gcd(a_1, a_2, ..., a_{n-1}, a_n)$, l'intervalle d'équations (2.6) admet une solution entière si et seulement si $L \leq d\lceil L/d \rceil \leq U$.

Le I test élimine sucessivement chacune des variables du système et applique ce test du GCD modifié pour chacune des équations obtenues jusqu'à ce qu'une réponse négative soit obtenue (ce qui atteste de l'indépendance) ou que toutes les variables soient éliminées (ce qui atteste d'une dépendance).

2.2.2 Linéarisation

La combinaison linéaire d'un ensemble d'équations (2.1) permet l'obtention d'une nouvelle équation :

$$\sum_{i=1}^{d} f_i\ (I_s) \prod_{j=i+1}^{d} N_j = \sum_{i=1}^{d} g_i\ (I_t) \prod_{j=i+1}^{d} N_j \qquad (2.8)$$

Le test de dépendance basé sur la résolution de cette nouvelle équation de linéarisation a été introduit dans [BuCy86] [GiPo88].

3.
$$a^+ = \begin{cases} a,\ si\ a \geq 0 \\ 0,\ sinon \end{cases} \qquad a^- = \begin{cases} -a,\ si\ a \leq 0 \\ 0,\ sinon \end{cases}$$

Toute solution de l'ensemble d'équations (2.1) est aussi solution de l'équation de linéarisation (2.8). Par contre, l'équation (2.8) admet des solutions entières qui ne sont pas solutions de l'ensemble des équations de (2.1). Prenons, pour exemple, un tableau $A(1:20, 1:10)$ et deux références au tableau $A(i_1, j_1)$ et $A(i_2, j_2)$. L'équation de linéarisation $i_1 - i_2 = 10 * (j_2 - j_1)$ conduit à plusieurs solutions dont $i_1 = 11, i_2 = 1, j_1 = 1, j_2 = 2$, qui n'est pas solution de l'ensemble d'équations $\{i_1 = i_2, j_1 = j_2\}$.

Le système de dépendance obtenu en remplacant (2.1) par (2.8) est noté DSL. Nous avons les relations suivantes :

$S_{int}(DS) \subseteq S_{int}(DSL)$ et $S_{int}(DS) \subseteq S_{reel}(DS) \subseteq S_{reel}(DSL)$.

L'objective de la linéarisation est de transformer un problème de résolution d'un ensemble d'équations en un problème de la résolution d'une seule équation. Les tests mono-dimensionnels présentés ci-dessus peuvent être utilisés pour détecter la présence d'une solution au système linéarisé DSL. Cette méthode de la linéarisation a été utilisée dans le paralléliseur **PTRAN** développé au centre Watson d'IBM [ABCC87].

2.2.3 Solutions entières sans contraintes

Ce type de méthode vise à chercher les solutions entières qui satisfont un ensemble d'équations (2.1) tout en ignorant les bornes du domaine d'itérations ($S_{int}\ (Eq)$). Puisque l'ensemble d'équations Eq est un sous-ensemble des contraintes du système DS, l'ensemble des solutions entières du système DS est inclus dans celui des solutions entières du système d'équations Eq, nous avons la relation $S_{int}(DS) \subseteq S_{int}(Eq)$. Nous présentons maintenant deux exemples de ce type de test : le test GCD généralisé de Banerjee et le Δ test.

GCD généralisé de Banerjee [Bane88] :

Le test du GCD généralisé est une extension du test du GCD pour un système d'équations. Soit $AX = C$ l'ensemble des équations (2.1) (Fig2.2) sous forme matricielle; avec $X = (x_1, x_2, ..., x_n)$, $C = (c_1, c_2, ..., c_d)$, et A une matrice entière de dimension $n \times d$. En appliquant un ensemble de transformations élémentaires à A, il est toujours possible de trouver la ma-

trice triangulaire H de Hermite associée à A vérifiant $AU = H$ où U est une matrice unimodulaire $(n \times n)$ [Schr86]. **le système $AX = C$ admet une solution entière X si et seulement s'il existe une solution entière $T = (t_1, t_2, ..., t_n)$ au système d'équations $HT = C$** [Bane88]. La matrice H étant triangulaire, ce système est plus facile à résoudre que le système d'équations initial (la phase de remontée de l'élimination de Gauss peut être par exemple utilisée). Si ce système n'admet aucune solution, il y a pas de solution entière pour $AX = C$ (i.e. $S_{int}\ (Eq) = \emptyset$), ce qui atteste qu'il n'y a pas de dépendance; sinon, la solution entière X s'exprime $X = UT$.

Le test du GCD généralisé a été adopté par de nombreux prototypes tels que : **PARAFRASE-2** [PePa91], **SUIF** [MHL91], **TINY** [WoTs91].

Le Δ test [GKT91] :

Le Δ test est un test multi-dimensionnel utilisé par les deux paralléliseurs de Rice : **PFC** et **ParaScope**. Le test de dépendance, qu'ils utilisent, partitionne les indices du tableau référencés en deux catégories :

- ceux dont les variables ne figurent dans aucun autre indice (le premier indice du tableau pour les références $A(i, k, k+1)$ et $A(j, k, k+2)$ en est un exemple)
- des autres cas (indice référençant au moins une variable présente au sein de plusieurs indices, comme les indices 2 et 3 de l'exemple précédent).

Ces indices sont ensuite classés selon trois types (fonction du nombre de variables qu'ils contiennent) : ZIV (Zero Index Variable), SIV (Single Index Variable), MIV (Multiple Index Variable). Pour les indices "libres", *le test exact* et *le test de Banerjee* sont utilisés; *le Δ test* est appliqué dans les autres cas.

Le Δ test ajoute aux équations les variables de distance de dépendance et les contraintes sur les variables de distance correspondantes dans le système de dépendance ($d_1 = i' - i \quad et \quad d_2 = j' - j$, par exemple, dans le cas de

deux boucles imbriquées). Ensuite, il effectue récursivement les trois phases suivantes :

1. tests exacts pour les indices de types ZIV ou SIV;
2. intersection des équations portant sur les même variables afin de chercher les contradictions ou d'éliminer les contraintes rédondantes;
3. propagation des valeurs des variables de distance *connues* dans le système de contraintes. Par exemple, l'équation $d_1 = 1$ peut être propagagée dans le système $d_1 + d_2 = 0$, on obtient la nouvelle contrainte $d_2 = -1$.

Le Δ test est plus puissant que le test du GCD car il permet parfois de conclure à l'inexistence d'une solution (indépendance) au système (réponse du test du GCD sur la contrainte générée) tandis que le test du GCD sur chacune des équations ne permet pas de conclure. Le Δ test est moins précis que le GCD généralisé ou que d'autres algorithmes procédant à l'élimination exacte des équations (ex. l'*Omega* test), parce qu'il n'est pas toujours possible de résoudre un système d'équations par propagations des contraintes. Si le Δ test ne permet pas de conclure à l'inexistence de solution pour le système d'équations, cela ne garantit donc pas l'existence d'une solution entière pour autant.

2.2.4 Solutions réelles avec contraintes

Ce type de méthode teste l'existence de solution réelle vérifiant tout le système de contraintes. L'ensemble des solutions donne une approximation de l'ensemble des solutions entières recherchées.

Lorsqu'on relaxe la condition de solution *entière*, le problème de programmation en nombres entiers n'est plus qu'un problème de programmation linéaire. Les algorithmes traditionnels de programmation linéaire tels que : la *méthode du simplexe* [Schr86] et la *méthode de loop residue* [Shos81] permettent de résoudre ce type de problème. Cependant, ces algorithmes ne conviennent pas très bien au problème du test de dépendance car il apporte une solution exacte au problème

alors que l'existence (ou l'inexistence) d'une solution est suffisante. Nous présentons donc maintenant des algorithmes moins complexes qui sont plus appropriés au test de dépendance.

Elimination d'une variable par l'algorithme de Fourier-Motzkin :

Initialement l'algorithme de Fourier-Motzkin permettait de rechercher une solution réelle à un système d'inégalités [DaEa73] [Duff74]. Il a été étendu pour tester la faisabilité d'un système de dépendance contenant des équations [Kuhn80] [TIF86].

Soit un système linéaire d'inégalités sous la forme (2.9)

$$\sum_{j=1}^{n} a_{ij}x_j \ \geq \ b_i, \ i = (1, ..., m) \qquad (2.9)$$

La méthode de Fourier-Motzkin procède par éliminations successives des variables du système par combinaisons linéaires de paire d'inégalités. Nous détaillons maintenant ce processus (élimination de la variable x_1) :

1. le système (2.9) est découpé en trois parties selon le signe des coefficients de x_1.

$$Pos = \begin{cases} a_1x_1 \ \geq \ D_1(\bar{x}) \\ . \\ . \\ . \\ a_px_1 \ \geq \ D_p(\bar{x}) \\ \\ (\forall i \in 1..p, \ a_i > 0) \end{cases} \qquad Neg = \begin{cases} a_1'x_1 \ \leq \ E_1(\bar{x}) \\ . \\ . \\ . \\ a_q'x_1 \ \leq \ E_q(\bar{x}) \\ \\ (\forall i \in 1..q, \ a_i' > 0) \end{cases}$$

$$Zero = \begin{cases} 0 \ \leq \ F_1(\bar{x}) \\ . \\ . \\ . \\ 0 \ \leq \ F_r(\bar{x}) \end{cases}$$

où $\bar{x} = (x_2, ..., x_m)$

2. la variable x_1 est éliminée par combinaisons de paire d'inéquations de Pos et Neg. Le nouveau système obtenu est le suivant :

$$\begin{cases} a_j' D_i(\bar{x}) \ \leq \ a_i E_j(\bar{x}), i = (1, ..., p), j = (1, ..., q) \\ \\ 0 \ \leq \ F_k(\bar{x}), k = (1, ..., r) \end{cases}$$

Si au cours du processus d'élimination des variables 1) une contrainte infaisable est détectée (ex. 0 $\leq$ -b, b$\in$N), le système est déclaré non faisable 2) le système des contraintes est vide , le système est déclaré faisable [DaEa73].

Théoriquement, cet algorithme est de complexité exponentielle. Cependant, d'après nos expériences (voir 2.5), dans la pratique, il reste de complexité polynômiale.

Ce test est utilisé par de nombreux prototypes de parallélisation automatique tels que: **SUIF** [MHL91], **TINY** [WoTs91], **PIPS** [IJT91]. Le test de **PIPS** est *renforcé* par une *heuristique* permettant de tronquer les inégalités par le PGCD des coefficients. C'est une heuristique parce que le résultat dépend de l'ordre des éliminations des variables.

Le λ-test [LYZ89] [Grun90] :

Le λ-test est une version multi-dimensionnelle du test de Banerjee. Il permet de tester l'existence d'une solution réelle commune à un ensemble d'équations linéaires (2.10) dans un domaine (2.11) défini numériquement (l_i, u_i $(1 \leq i \leq n)$ sont constantes). Il permet donc de calculer $S_{reel}(DS)$.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n + c_1 = 0 \quad (f_1) \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n + c_2 = 0 \quad (f_2) \\ ..... \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n + c_m = 0 \quad (f_m) \end{cases} \tag{2.10}$$

$$\begin{cases} l_1 \leq x_1 \leq u_1 \\ l_2 \leq x_2 \leq u_2 \\ ... \\ l_n \leq x_n \leq u_n \end{cases} \tag{2.11}$$

L'idée principale est que toute solution S de (2.10) est aussi solution de l'ensemble des combinaisons linéaires des équations (2.12).

$$\lambda_1 f_1 + \lambda_2 f_2 + ... + \lambda_m f_m = 0 \quad (f) \tag{2.12}$$

Pour garantir que S est inclus dans le polyèdre V défini par (2.11), il faut que toutes les solutions de (2.12) intersectent V. Autrement dit, (2.10) admet une solution réelle dans V si et seulement s'il n'existe pas d'hyperplan f de (2.12), tel que l'intersection de f avec V soit vide.

Ce test a été implémenté dans **PARAFRASE**. D'après [LYZ89], le λ-test a la même précision que la méthode de Fourier-Motzkin mais est de complexité inférieure. Pour effectuer une comparaison plus précise, une évaluation de leurs complexités moyennes dans la pratique serait nécessaire. D'après nos expériences, la complexité réelle de l'algorithme de Fourier-Motzkin dans le cadre d'un test de dépendance, reste en moyenne polynômiale (voir la section 2.5).

Le test de Fourier-Motzkin traite des cas plus généraux et permet de projeter le système de dépendance sur le sous-espace D des variables de distance de dépendance. Les différentes approximations des dépendances résumant D peuvent être calculées à partir ce système, qui est beaucoup plus petit que le système initial (voir l'algorithme 2.2/2.3 dans la suite de ce chapitre et le calcul du cône de dépendance dans le chapitre 3).

2.2.5 Tests exacts

Les tests exacts testent l'existence de solution entière pour le système de contraintes complet représenté en 2.2 (S_{int} (DS)).

Ce type de problème de Programmation Linéaire en Nombres Entiers peut être résolu en utilisant les algorithmes classiques tels que : la méthode des coupes de Gomory [Gree71] ou une extension de l'algorithme de Fourier-Motzkin [Will76]. Nous présentons dans cette section des algorithmes exacts utilisés tout particulièrement pour le test de dépendance.

PIP (Parametric Integer Programming) [Feau88] :

L'algorithme PIP est un algorithme de PE tout particulièrement inté-

ressant pour résoudre le problème du test de dépendance. C'est une une extension de la méthode des coupes de Gomory. Il calcule la solution entière lexicographique minimale du système en fonction de paramètres symboliques. L'utilisation de cet algorithme permet de calculer, pour chaque référence au tableau en lecture, la référence au tableau en écriture qui a produit la valeur de la variable. Le résultat se traduit sous la forme d'un ensemble d'égalités conditionnées par des prédicats [Feau89] [Feau91]. Cet algorithme est utilisé par le paralléliseur PAF dans le calcul du *data-flow graph* qui est une version plus précise du graphe de dépendance.

Extension du GCD généralisé [Bane88] :

L'idée essentielle de cette méthode est d'exprimer la solution entière des variables apparaissant au sein des équations du système en fonction de paramètres entiers, puis de substituer ces variables dans l'ensemble des inégalités par leur expressions paramètrées, avant de chercher les valeurs possibles de tous ces paramètres.

Rappelons que l'ensemble des équations $AX = C$ admet une solution entière si et seulement s'il existe un vecteur T tel que : $HT = C$, où H est la forme de Hermite [Schr86] associée à A et $AU = H$. La solution s'exprime $X = UT$.

Supposons que le rang de la matrice A est r. Il est possible de calculer les solutions numériques pour r variables de T. La solution générale de X aura $n - r$ paramètres libres. Si $r = n$, il existe une solution exacte unique $X = UT$. Sinon, après élimination de toutes les équations, le problème se traduit sous la forme d'un système en nombres entiers d'inégalités ayant $n - r$ variables . Lorsque ce système d'inégalités est soluble exactement, le système initial est aussi soluble exactement. Dans certains cas particuliers tels que :

- $r = n - 1$,
- il ne reste qu'une variable par contrainte,
- tous les coefficients de toutes les variables sont égaux à 1 en valeur

absolue,

le système d'inégalités peut être résolu *exactement* en nombres entiers par des algorithmes classiques de résolution de systèmes linéaires *rationnels*.

Le paralléliseur **SUIF** utilise cette méthode [MHL91]. Il transforme un système d'égalités et d'inégalités en un système d'inégalités en utilisant l'extension du GCD généralisé. Il utilise, ensuite, une série de tests exacts sous certaines conditions, pour résoudre le système d'inégalités.

L'*Omega* test [Pugh92] :

L'*Omega* test est un test exact basé sur l'algorithme d'élimination d'une variable de Fourier-Motzkin [Will76]. L'originalité de ce test porte sur sa technique de résolution en entiers du système d'inégalités.

Supposons que l'on cherche à résoudre le système d'inégalités $P(v_1, v_2, ..., v_n)$. Après élimination d'une variable v_k par l'algorithme de Fourier-Motzkin, le problème est celui de la résolution en nombres entiers du système $P'(v_1, ...v_{k-1}, v_{k+1}, ..., v_n)$. P' est une condition *nécessaire* de P, c'est à dire P *n'a pas de solution entière si P' ne possède pas de solution entière* [DaEa73]. L'*Omega* test utilise une version révisée de l'algorithme d'élimination d'une variable : la variable est éliminée du système P de manière à obtenir un système *suffisant* $P''(v_1, ...v_{k-1}, v_{k+1}, ..., v_n)$ tel que l'existence d'une solution entière à P'' garantit l'existence d'une solution entière pour P. Pour traiter les cas où P'' n'admet pas de solution entière alors que P en admet au moins une, une phase supplémentaire est utilisée.

Puisque l'*Omega* test est une extension de l'algorithme de Fourier-Motzkin, sa complexcité théorique est aussi exponentielle. Toutefois, pour les cas pratiques, sa complexcité est bornée par un temps polynomial [Pugh92].

Cet algorithme a été implémenté dans les prototypes **TINY** [Pugh92] et **PARAFRASE** [PePa92b].

2.2.6 Exemples de quelques paralléliseurs

Pour augmenter la précision de leur test de dépendance, les paralléliseurs utilisent généralement plusieurs algorithmes de recherche de solution au système caractérisant les dépendances. Ils commencent par des tests simples et peu coûteux pour terminer avec des algorithmes de précision meilleure. Nous présentons ici les choix qui ont été effectués lors de l'implémentation des tests de dépendances de quelques paralléliseurs.

- **PAF** [TDF87] (Paralléliseur Automatique de Fortran) développé au laboratoire MASI utilise l'algorithme PIP (voir la section 2.2.5) qui est un algorithme de résolution de systèmes d'équations et d'inéquations paramétriques en nombres entiers [Feau88].

- **PARAFRASE** développé à l'Université de l'Illinois utilise successivement *le test du GCD*, *GCD généralisé*, *le test de Banerjee* et *un test de la programmation en nombre entière*[4] [PePa91].

- **ParaScope** ou **PFC** développé à l'Université de Rice distribue le système de dépendance en plusieurs sous-systèmes, correspondant aux différentes dimensions de la fonction d'accès aux éléments du tableau et sous la condition que leurs intersections soient disjointes. Des tests de dépendance individuels sont effectués sur ces sous-systèmes indépendants. Pour les systèmes où un seul indice du tableau intervient, *le test exact* et *le test de Banerjee* sont utilisés; *le* Δ *test* (voir la section 2.2.3) est appliqué sur les autres systèmes [GKT91].

- **PIPS** un paralléliseur développé à l'Ecole des Mines utilise : une variante du *test du GCD généralisé* et *l'algorithme de Fourier-Motzkin* [IJT91]. Nous détaillerons cet algorithme dans la section 3.3.3.

- **PTRAN** développé au centre de recherche Watson d'IBM utilise *le test du GCD* et *le test de Banerjee-Wolfe* avec l'équation de linéarisation [ABCC87].

4. ce test a été remplacé par l'*Omega* test [PePa92b]

- **SUIF** développé à Stanford effectue d'abord *le test GCD généralisé*. Ensuite, il applique 4 tests de résolution d'un système d'inéquations, dans l'ordre : *le test de contrainte sur les variables simples, le test acyclique, le test* simple loop residue*, l'algorithme de Fourier-Motzkin* [MHL91].

- **TINY** un environement de transformations de programme, développé à *Oregon Graduate Institute*, a adopté *Power test* [WoTs91] qui est une combinaison de l'algorithme du GCD Généralisé et de la méthode de Fourier-Motzkin. L'*Omega* test a été intégré récemment [Pugh92].

2.3 Analyse de l'exactitude

Rappelons que les tests approximatifs sont pessimistes car l'existence d'un seul élément, même réel, dans l'ensemble des solutions conduit à la possibilité d'existence d'une dépendance. Une dépendance déduite d'un test approximatif est cependant réelle si l'existence de solution dans l'ensemble approximé implique l'existence de solution entière. Analyser les conditions suffisantes où les tests approximatifs donnent des résultats exacts permet de distinguer les *vraies* dépendances des *fausses*.

2.3.1 Conditions suffisantes

Les tests approximatifs, permettant de trouver des solutions entières ne tenant pas compte des inégalités S_{int} (Eq) ou des solutions réelles prenant en considération toutes les contraintes S_{reel} (DS), sont ceux les plus souvent utilisés dans les paralléliseurs. Nous détaillons dans cette section le type de résultat (exact ou non) auquel ils peuvent conduire.

- Pour une mono-équation du type : $a_1 * i_1 + a_2 * i_2 + ... + a_n * i_n = a_0$ où tous les coefficients non-nuls valent 1 en valeur absolue et où les bornes des indices i_j sont constantes, le test de Banerjee est exact [Bane76].

- Pour une mono-équation du type : $a_1 * i_1 + a_2 * i_2 + ... + a_n * i_n = a_0$ où un au moins des coefficients vaut 1 en valeur absolue et où les bornes des

indices i_j sont constantes, [LiYe89] et [KPK90] proposent deux conditions certifiant que l'existence de solution réelle implique l'existence de solution entière avec les contraintes.

- Pour un couple d'équations dont les coefficients non-nuls valent 1 ou -1 et pour des contraintes constants, il est possible de savoir si ce système d'équations admet des solutions entières en combinant deux tests : celui testant l'existence de solution entière sans contraintes S_{int} (Eq) et celui testant l'existence de solution réelle avec contraintes S_{reel} (DS) [LiYe89]. Les deux tests du GCD généralisé et le λ-test peuvent donc être utilisés pour l'obtention d'un résultat exact.

- Pour les autres cas, [Anco91] propose trois conditions suffisantes permettant de savoir si l'élimination d'une variable par l'algorithme de Fourier-Motzkin conduit à un résultat exact (projection entière). Ce sont les trois conditions suivantes :

  Soit

  $$S \;=\; \begin{cases} (A_1 - c_1) \;+\; d_1\; k \;\leq\; 0 \\ (A_2 - c_2) \;-\; d_2\; k \;\leq\; 0 \end{cases}$$

  le système linéaire caractérisant un ensemble points entiers IP_s.

  et

  $$SP \;=\; \left\{ d_1\; (A_2 - c_2) \;\leq\; d_2\; (-A_1 + c_1) \right.$$

  le système linéaire résultant de l'élimination de la variable k des deux premières inégalités du système S , en utilisant la méthode de Fourier-Motzkin; caractérisant un ensemble points entiers IP_{sp}.

  La projection de IP_s selon la variable k est égale à IP_{sp}, si au moins une des trois conditions suivantes est vérifiée :

  1. le coefficient $d_1 \;=\; 1$
  2. le coefficient $d_2 \;=\; 1$

3. $d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1 \leq -d_2\ (A_1 - c_1)$ est redondante pour le système S

L'existence de solution entière dans le polyèdre initial est équivalent à l'existence de solution entière dans le polyèdre après l'élimination de la variable k si toutes les paires d'inéquations contenant la variables k vérifient au moins l'une de ces trois contitions.

Une condition toujours suffisante mais moins restrictive a été introduite dans [Pugh92]. La troisième condition peut être remplacée par :

- $d_1\ (A_2 - c_2)\ +\ d_1\ d_2\ -\ d_1\ -\ d_2 + 1 \leq -d_2\ (A_1 - c_1)$ est redondante pour le système S

Cette condition est à la base de l'*Omega* test.

- Si après utilisation de l'algorithme de Fourier-Motzkin, pour l'élimination des variables du système de manière exacte, il reste encore des contraintes, D.E. Maydan, J.L. Hennessy et M.S. Lam proposent la méthode suivante. Une solution *simple* est calculée par substitution arrière [MHL91] des variables du système. Si cette solution est entière, l'existence d'une solution entière au système est vérifiée. Sinon ils suggèrent l'utilisation d'une méthode de type *branch and bound* pour tester l'existence d'une telle solution (Ce cas ne s'est jamais produit [MHL91] sur l'ensemble des tests qu'ils ont effectués).

2.3.2 Utilité de l'exactitude

L'étude empirique effectuée par Shen & Li [SLY89], sur les tests de dépendance et le format des indices de tableau présents dans les applications numériques, montre que les coefficients des indices des tableaux valent fréquemment 1 ou -1. Parmi 4105 paires de références bi-dimensionelles, 97.37% vérifient cette condition. Ceci est donc en faveur de l'utilisation des tests $S_{reel}\ (DS)$ et $S_{int}\ (Eq)$ plutôt que celle des tests exacts $S_{int}\ (DS)$ plus coûteux et rarement nécessaires. On définit l'exactitude d'un test approximatif par le rapport du nombre de réponses exactes (indépendances et dépendances exactes) et celui des tests effectués.

L'exactitude du test de dépendance développé dans **PIPS** sur le benchmark PerfectClub est de 97.95% (évaluation faite avec les conditions de C. Ancourt (voir la section 2.5)). Les expériences de SUIF aussi sur le PerfectClub, donnent une exactitude de 100% [MHL91] (la méthode présentée précédemment de D.E. Maydan, J.L. Hennessy et M.S. Lamdu étant utilisé par le test de dépendance).

2.4 Test de dépendance de 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 programmes FORTRAN 90 ou FORTRAN parallèle. Les boucles DO séquentielles parallélisables sont remplacées par des instructions vectorielles FORTRAN 90 ou par des constructions de type DOALL. Les trois caractéristiques principales de **PIPS** sont :

- *l'interprocéduralité* qui est prise en compte dans les phases d'analyse du programme, de calcul du test de dépendance, et de parallélisation;
- *une analyse sémantique sophistiquée*, basée sur les préconditions et les régions, qui permet l'obtention de résultats très précis lors du calcul des effets, du test de dépendance, 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.

L'algorithme du test de dépendance [TIF86] de **PIPS** peut utiliser les résultats des phases de l'analyse interprocédurale (les *régions*) et de l'analyse sémantique (les *prédicats*). La présence de constante symbolique dans les expressions des bornes de boucles (ex. $DO\ I = 1, N$) ou dans les références aux éléments des tableaux (ex. $T(I + 2 * N)$) ne restreint pas l'efficacité du test, tant qu'elle ne conduit pas à une forme non linéaire (ex. I*N). Ces deux phases d'analyse du programme dont les résultats vont influencer l'efficacité du test de dépendance sont présentées ci-dessous.

Nous détaillons, dans cette section, les différentes étapes de **PIPS** qui interviennent dans le calcul du graphe de dépendances, avant de présenter le test de dépendance et ses caractéristiques. Nous terminons notre présentation par les résultats expérimentaux que nous avons obtenus et les améliorations introduites.

2.4.1 Calcul de *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 initial. 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 ne sont pas pris en compte (des arcs entre deux éléments $T(2)$ et $T(3)$ peuvent exister). Une phase d'analyse des dépendances est donc 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.

2.4.2 Analyse sémantique

Le système de dépendance classique (voir section 2.1.2) est constitué de deux ensembles : un ensemble d'égalités caractérisant une dépendance possible entre deux références et un ensemble d'inéquations caractérisant le domaine d'itérations. Ce système est construit à partir d'un nid de boucles où : (1) les boucles sont normalisées; (2) les expressions d'indices des tableaux sont tous des fonctions des indices des boucles; (3) il n'y a pas de test IF ou d'appel de procédure CALL dans le corps de boucles. Sous ces conditions, le système représenté par la figure 2.2 correspond à l'image réelle du problème de dépendance. Dans les autres cas, des contraintes supplémentaires doivent être ajoutées dans le système pour ne pas perdre de précision. Dans la pratique, les constantes symboliques étant souvent présentes dans les fonctions d'accès aux éléments des tableaux (50%) et dans les bornes de boucles (95%) [HaPo90], la phase d'analyse sémantique apporte des

informations importantes pour la phase de parallélisation.

Prenons l'exemple suivant :

```
SUBROUTINE exemple1(A, N, M)
REAL A(N+M)

IF (M.GE.N) THEN
  DO I = 1, N
    A(I) = A(I+M)
  ENDDO
ENDIF
END
```

FIG. 2.3 - *Exemple 2.1*

Le système de dépendance entre les deux références au tableau `A` : `A(I)` et `A(I+M)` peut se calculer soit en tenant compte des prédicats, soit sans ces prédicats.

- Soit $S1$ le système constitué de l'équation sur les indices du tableau et des contraintes sur les bornes de boucle :

$$S1 : \quad \begin{cases} i = i' + M \\ 1 \leq i \leq N \\ 1 \leq i' \leq N \end{cases}$$

  Les variables symboliques M, N sont des variables entières non contraintes. Ce système $S1$ possède des solutions entières et on en déduit l'existence d'une dépendance.

- Soit $S2$ le système plus complet contenant des contraintes sur M et N, résultant des prédicats :

$$S2 : \quad \begin{cases} i = i' + M \\ 1 \leq i \leq N \\ 1 \leq i' \leq N \\ N \leq M \end{cases}$$

Le système $S2$ est infaisable. En tenant compte des prédicats vérifiés pas les variables symboliques du programme, de fausses dépendances entre les éléments du tableau peuvent être éliminées.

L'utilisation d'information supplémentaire portant sur les variables symboliques du programme est importante puisqu'elle peut permettre d'éliminer de *fausses* dépendances. Ce type d'information, connu à la compilation, résulte de la phase d'analyse sémantique.

La phase d'analyse sémantique de **PIPS** génère, pour toutes les instructions du programme, les prédicats vérifiés par les variables scalaires avant l'exécution de chacune des instructions [IJT91]. Ces prédicats sont représentés par des polyèdres (système d'égalités et d'inégalités) . Les polyèdres représentent une formulation suffisamment générale pour exprimer les informations extraites des différentes analyses classiques telles que : la propagation de constante, expression de variables inductives, égalités linéaires entre variables, contraintes linéaires générales,... [CoHa78]. Cette phase est basée sur la méthode développée par P. Cousot et N. Halbwachs [CoHa78] pour l'analyse intra-procédurale des programmes. Cette méthode a été étendue pour prendre en considération certains problèmes dûs à l'aliasing (explicite créé par la déclaration EQUIVALENCE) et pour tenir compte de l'inter-procédural.

2.4.3 Analyse des effets des procédures : SDFI et Région

L'analyse interprocédurale de **PIPS** permet le calcul des préconditions et des effets interprocéduraux. Deux types d'effets sont évalués : le SDFI (*Summary data flow information*) et les régions.

Le SDFI représente les effets d'une procédure sur les variables non statiques locales modifiées ou lues par la procédure. Un résumé de cette information est utilisé pour calculer les effets d'une instruction CALL sur les variables du programme. La caractéristique du SDFI implémenté dans Pips se situe au niveau de la précision des éléments de tableau modifiés par la procédure. Plus précisément, si les paramètres réels et formels de la procédure sont des tableaux de tailles différentes, des informations comme l'intervalle des éléments du tableau modifiés

seront utilisées pour indiquer, par exemple, qu'une colonne d'une matrice a été modifiée. L'écriture de la J-ème colonne d'une matrice $X(N \times N)$ sera représentée par l'effet :

$$< MUST\ BE\ WRITTEN >: X((/I, I = 1, N, 1/), J)$$

Une région, telle qu'elle est définie dans [TIF86], est un ensemble d'éléments de tableau que l'on peut représenter par un polyèdre (système d'égalités et d'inégalités). Elle caractérise la partie du tableau manipulée par une procédure. La région représentant l'effet sur un tableau T d'une procédure :

```
IF (M. EQ. 1)
   DO I = 1, N
       T(2*I+M) = 1
   ENDDO
ENDIF
```

est $(T(\phi_1),\ \{\phi_1 = 2I + M, 1 \leq I \leq N, M = 1\})$. Les régions sont calculées intra- et inter-procéduralement et représentent les éléments de tableau référencés soit par des instructions élémentaires soit par des instructions composées (ex. test, loop, ...). La "région d'une procédure" résume les régions associées à chaque instruction du corps de procédure; les effets locaux n'interviennent pas dans ce résumé. L'utilisation des résultats de l'analyse sémantique interprocédurale permet de raffiner cette information [Irig92].

L'algorithme du calcul des régions proposé par R. Triolet & al. [TIF86] a été implémenté dans **PIPS** [Plat90].

2.4.4 Construction d'un système de dépendance

Rappelons que le problème du test de dépendance est de résoudre en entiers *un système linéaire* qui caractérise une dépendance possible entre des éléments référencés. Le domaine d'itérations correspond au domaine *explicite* caractérisant les variables. Les prédicats vérifiés par les variable scalaires et les indices de boucles font partie du domaine *implicite*.

Un des objectifs de cette thèse est d'étudier l'intérêt d'une phase d'analyse sémantique approfondie pour la détection des dépendances et la phase de parallélisation. Afin de pouvoir effectuer des comparaisons, nous distinguons trois types de système de dépendance.

Le *système simple* qui contient :

1. l'ensemble des équations définissant une dépendance possible entre deux références du tableau ((2.1) dans la Fig 2.2).

Le *système normal* qui contient :

1. l'ensemble des équations définissant une dépendance possible entre deux références du tableau ((2.1) dans la Fig 2.2)
2. l'ensemble des inéquations caractérisant le domaine d'itérations ((2.2) dans la Fig 2.2).

Le *système complet* qui contient :

1. l'ensemble des équations définissant une dépendance possible entre deux références du tableau ((2.1) dans la Fig 2.2)
2. l'ensemble des inéquations caractérisant le domaine d'itérations ((2.2) dans la Fig 2.2).
3. l'ensemble des prédicats (égalités et inégalites) vérifiés par les variables scalaires et indices de boucle obtenus au cours de l'analyse sémantique.

Trois analyses différentes[5] des dépendances, associées au différent système, ont été implémentées dans **PIPS**, car si la parallélisation efficace de certains programmes nécessite l'utilisation de techniques sophistiquées (faisant appel à de nombreuses informations), d'autres n'en ont pas besoin.

Reprenons l'exemple 2.1 et testons l'existence d'une dépendance entre `A(I)` et `A(I+M)`. Les trois systèmes précédents se traduisent :

– Système simple :

$$i = i + di + M$$

5. Quatre si on prend en compte l'analyse de dépendances fondée sur les régions

équivalent après simplification à :

$$di + M = 0$$

– Système normal :

$$\begin{cases} di + M = 0 \\ 1 \leq i \leq N \\ 1 \leq i + di \leq N \end{cases}$$

– Système complet :

$$\begin{cases} di + M = 0 \\ 1 \leq i \leq N \\ 1 \leq i + di \leq N \\ N \leq M \end{cases}$$

La variable di est la variable de distance des dépendances définie par $i' - i = di$. Nous l'utilisons dans certains de nos exemples pour des raisons de simplicité, i' est alors remplacé par $i + di$ dans le système de dépendance.

Dans cet exemple, seule l'analyse complète permet l'obtention d'un résultat *exact*: l'indépendance. Les expériences effectuées sur ces trois types d'analyse des dépendances montrent que l'amélioration apportée par l'utilisation du *test complet* par rapport au *test normal* est identique à celle apportée par le *test normal* par rapport au *test simple*. Nous détaillons ces résultats dans la section 2.5.

2.4.5 Algorithme du test de dépendance

Lorsque le système de dépendance est construit, le test de dépendance est appliqué. Il teste la faisabilité du système et calcule les abstractions de dépendance : *profondeurs de dépendance, cône de dépendance* ... qui seront utilisées au cours de la parallélisation et de l'optimisation des programmes.

Cet algorithme applique successivement les tests de dépendance suivant, du plus simple au plus complexe : *le test des constantes, le test du GCD, le pre-test de lexico-positivité, le test de consistance, le test de faisabilité, et le test de lexico-positivité.*

1. **Test des constantes**

   Une fonction d'indice du tableau est "constante" si elle est invariante au

cours de l'exécution du programme. Cette fonction peut être une valeur numérique ou un terme symbolique. Le *test des constantes* compare chaque paire d'indices constants du tableau. Si cette paire d'indices est différente, les deux références sont indépendantes. L'exemple ci-dessous illustre ce cas :

```
DO I = 1, N
    DO J = 1, N
      A(L,I,J) = A(L+1,I,J-1)
    ENDDO
ENDDO
```

Considérons les deux références à `T` : `A(L,I,J)` et `A(L+1,I,J-1)`. L et $L+1$ étant des constantes différentes, les deux références ne peuvent être qu'indépendantes.

2. **Test du GCD**

   Le test du GCD s'applique à une équation diophantienne. Il vérifie que cette équation admet une solution entière en testant si le plus grand commun diviseur des coefficients de l'équation divise bien le terme constant. L'exemple suivant illustre l'utilité de ce test.

```
DO I = 1, N
     A(2I) = A(2I-1)
ENDDO
```

   En prenant le système simple de dépendance des références `A(2I)` et `A(2I-1)`, on obtient l'équation :

$$2i = 2i' - 1$$

   Comme GCD(2,2) ne divise pas 1, elles sont indépendantes.

3. **Pre-test de lexico-positivité**

   Ce test assure que la condition 3 du système de dépendance (ref. 2.1.2), se rapportant à l'ordre d'exécution des instructions, est bien vérifiée. Il est appliqué sur le système initial lorsque les distances de dépendance sont constantes. Le système est déclaré *indépendant/infaisable* si le vecteur de distances $(d_1, ..., d_n)$ est constant et lexico-négatif. Le test de dépendance

prend fin dès que ce test est négatif et aucune vérification de consistance du système complet ne sera nécessaire. Donnons un exemple :

```
DO I = 1, N
   DO J = 1, N
      A(I,I) = A(I+1,J)
   ENDDO
ENDDO
```

Considérons le système simple de dépendance de `A(I,I)` et `A(I+1,J)`.

$$\begin{cases} \text{i} = \text{i} + \text{di} + 1 \\ \text{i} = \text{j} + \text{dj} \end{cases} \quad \Longrightarrow \quad \begin{cases} \text{di} = \text{-1} \\ \text{i - j - dj} = 0 \end{cases}$$

La première équation atteste que le vecteur de distance doit avoir la forme (-1, *), et qu'il est lexico-négatif. Aucune solution de ce type n'étant prise en compte, le résultat du test est négatif, ce qui entraine l'arrêt du test de dépendance.

4. **Test de consistance d'un système**

   Un système est non consistant s'il représente un ensemble vide i.e. contient des contraintes contradictoires sur les variables. Dans **PIPS**, ce test de consistance est effectué lors de la normalisation du système de dépendance.

   La **normalisation d'un système** se divise en 2 étapes :

   - normalisation de chaque contrainte, c'est à dire division entière de chacun des coefficients par le PGCD des coefficients ;

     Soit la *contrainte*

     $$\sum_{i=1}^{n} a_i x_i \leq c \qquad ou \qquad \sum_{i=1}^{n} a_i x_i = c$$

     soit $k = pgcd(a_1, a_2, ..., a_n)$;

     Après normalisation, la *contrainte* est respectivement :

     $$\sum_{i=1}^{n} a_i x_i / k \leq c/k \qquad ou \qquad \sum_{i=1}^{n} a_i x_i / k = c/k$$

– Elimination de contraintes redondantes

(a) élimination des équations *identiques* redondantes :

$$\begin{cases} Ax - c = 0 \\ Ax - c = 0 \end{cases} \quad ou \quad \begin{cases} Ax - c = 0 \\ c - Ax = 0 \end{cases}$$

(b) élimination des inéquations redondantes dans les système du type :

$$\begin{cases} Ax \leq c \\ Ax \leq c \end{cases} \quad ou \quad \begin{cases} Ax = b \\ Ax \leq c \quad (avec \quad b \leq c) \end{cases} \quad ou \quad \begin{cases} Ax \leq b \\ Ax \leq c \end{cases}$$

(c) élimination des contraintes redondantes dans les système du type :

$$0 = 0 \qquad ou \qquad 0 \leq c \quad (avec \quad c \geq 0)$$

Le **test de consistance** associé à cette normalisation s'arrête dès que le système est déclaré *infaisable/indépendant* i.e. lorsque :

– une équation est détectée infaisable après application du test du GCD

– des contraintes contradictoires du type :

(a)

$$\begin{cases} Ax = b \\ Ax = c \end{cases} \qquad avec \quad (b \neq c)$$

(b)

$$\begin{cases} Ax \leq b \\ Ax = c \end{cases} \qquad avec \quad (b < c)$$

sont détectées

Nous illustrons ces cas sur l'exemple suivant :

```
DO I = 1, N
    DO J = 1, N
        A(I,I) = A(I,N+1) + A(J,J-1)
    ENDDO
ENDDO
```

– calcul du système de dépendance de `A(I,I)` vers `A(I,N+1)`.

$$\begin{cases} i = i + di \\ i = N + 1 \\ 1 \leq i \leq N \\ 1 \leq j \leq N \end{cases}$$

Les deux contraintes : $i = N + 1$ et $i \leq N$ sont contradictoires (cas (b)). Les deux références sont donc indépendantes.

– Calcul de la dépendance de `A(I,I)` vers `A(J,J-1)` en utilisant le système simple.

$$\begin{cases} i = j + dj \\ i = j + dj - 1 \end{cases} \quad \Longrightarrow \quad \begin{cases} i - j - dj = 0 \\ i - j - dj = -1 \end{cases}$$

Le système est encore *vide* car deux égalités sont contradictoires (cas (a)).

5. **Test de faisabilité d'un système**

Un système de contraintes linéaires est non faisable s'il n'admet pas de solution réelle. Le test de la faisabilité effectue des projections successives de toutes les variables du système jusqu'à ce que le système soit déclaré infaisable ou que toutes les variables aient été éliminées (système faisable). L'élimination des variables par propagation d'équation, contraignant la variable, sera effectuée avant l'élimination des variables par l'algorithme de Fourier-Motzkin, modifié par des troncatures des bornes, (exponentiel dans les pires cas). Le système initial est non faisable dès qu'un système (après élimination) est non consistant.

Nous détaillons ce test dans l'algorithme 2.1.

```
procedure TestFaisabilite(Sys, ListVars)
/* Sys est le système à tester;                                          */
/* ListVars est la liste des variables à éliminer dans Sys               */

if (TestConsist(Sys) = false)
        return(nonfaisable)

else
        Eq = Equations_of_Systeme(Sys)
        do until (Eq = empty)
                if Coefficient(v) est minimale dans l'équation eq
                        /* éliminer v dans Sys par l'élimination d'une équation  */
                        Sys = Projection_by_Equation(Sys, eq, v)
                        ListVars = ListVars - v
                        if (TestConsist(Sys) = false)
                                return(nonfaisable)
        enddo

        for toutes les variables v ∈ ListVars
                /* éliminer v dans Sys par l'algorithme de Fourier-Motzkin    */
               Sys = Projection_by_FourierMotzkin(Sys, v)
               ListVars = ListVars - v
               if (TestConsist(Sys) = false)
                       return(nonfaisable)
        endfor

endif
return(faisable)
end TestFaisabilite
```

FIG. 2.4 - *Algorithme 2.1. Test de la Faisabilité d'un Système linéaire*

Exemple :

```
DO I = 1, N
    DO J = 1, N
        A(I,I+1) = A(2J,2(I+J)) + A(1-J, I+1)
    ENDDO
ENDDO
```

Calculons respectivement les dépendances de `A(I,I+1)` vers `A(2J,2(I+J))` et `A(I,I+1)` vers `A(1-J, I+1)`.

- Le système *simple* de dépendance de `A(I,I+1)` vers `A(2J,2(I+J))`, $Sys(i,j,di,dj)$ est :

$$\begin{cases} i = 2(j+dj) \\ i+1 = 2(i+j+di+dj) \end{cases} \Longrightarrow \begin{cases} i-2j-2dj = 0 \\ i+2j+2di+2dj = 1 \end{cases}$$

  Après élimination de variable `i`, le système projeté $Sys(j,di,dj)$ est

$$4j + 2di + 4dj = 1$$

  Il est non faisable car $PGCD(4,2,4) = 2$ ne divise pas 1.

- Considérons le système *normal* de dépendance de `A(I,I+1)` vers `A(1-J, I+1)`, $Sys(i,j,di,dj)$:

$$\begin{cases} i = 1-(j+dj) \\ i+1 = i+di+1 \\ 1 \leq i \leq n \\ 1 \leq i+di \leq n \\ 1 \leq j \leq n \\ 1 \leq j+dj \leq n \end{cases} \Longrightarrow \begin{cases} i+j+dj = 1 \\ di = 0 \\ 1 \leq i \leq n \\ 1 \leq i+di \leq n \\ 1 \leq j \leq n \\ 1 \leq j+dj \leq n \end{cases}$$

  Après élimination de j, di par propagation des deux équations du système, et élimination de n, dj par l'algorithme de Fourier-Motzkin, le système projeté $Sys(i)$ est le suivant :

$$\begin{cases} -i \leq -1 \\ i \leq 0 \end{cases}$$

  Après élimination de i, on obtient une contrainte *non consistante :* $0 \leq -1$. Ce système est donc non faisable.

Nous en déduisons donc qu'il n'existe aucune dépendance entre les références au tableau A.

6. **Test de lexico-positivité**

   Ce test est plus complet que le *Pre-test de lexico-positivité* et est appliqué après *le test de faisabilité du système*. Il s'assure que la condition de lexico-positivité du vecteur des distances de dépendance est bien vérifiée, dans le cas général. Soit D le système caractérisant les distances de dépendance. Les bornes des valeurs prises par chacune des variables de distance de dépendance sont évaluées par projection de D sur la variable. Si le vecteur de distance est de la forme $(0, ..., d_i, ..., d_n)$ et que la borne supérieure de d_i est inférieure à 0, le test stoppe car le système est lexico-négatif et exprime qu'il y a indépendance.

   Voici un exemple :

```
DO I = 1, N
     DO J = 1, N
          A(I+J,J) = A(I+J+1,1) + 1
     ENDDO
ENDDO
```

   Recherchons l'existence d'une dépendance possible de `A(I+J,J)` vers `A(I+J+1,1)`. Le système de dépendance $Sys(i, j, di, dj)$ est le suivant :

$$\begin{cases} i + j = i + j + di + dj + 1 \\ j = 1 \\ 1 \leq i \leq n \\ 1 \leq i + di \leq n \\ 1 \leq j \leq n \\ 1 \leq j + dj \leq n \end{cases} \Longrightarrow \begin{cases} di + dj = -1 \\ j = 1 \\ 1 \leq i \leq n \\ 1 \leq i + di \leq n \\ 1 \leq j \leq n \\ 1 \leq j + dj \leq n \end{cases}$$

   Le *test de faisabilité* permet de vérifier que le système $Sys(i, j, di, dj)$ est bien faisable. Nous nous intéressons maintenant au sous-espace des variables de distance de dépendance $Sys(di, dj)$ défini par :

$$\begin{cases} di + dj = -1 \\ -dj \leq 0 \end{cases}$$

Si on élimine dj, on obtient une borne supérieure de di,

$$di \leq -1$$

qui est inférieure à 0. Nous en déduisons qu'il n'existe pas de dépendance entre `A(I+J,J)` vers `A(I+J+1,1)`.

L'algorithme d'analyse des dépendances (Algorithme 2.2) applique successivement ces 6 tests dans l'ordre de complexité croissante.

L'algorithme s'arrête dès que l'un de ces tests aboutit à un système traduisant l'indépendance. Si aucun de ces tests ne permet d'assurer que le système est *indépendant*, on considére qu'une dépendance existe. Les abstractions de cette dépendance sont alors calculées.

Dans cet algorithme, le test de la faisabilité est divisé en deux étapes, on projette tout d'abord les variables autres que les variables de distance, puis on élimine ces variables de distance du système après avoir conservé le système $SysD$ contraignant les variables de distance. Ce découpage permet de ne pas recalculer deux fois le système $SysD$.

```
procedure AnalyseDep(Sys)
/* Sys est le système de dépendance à analyser;                          */
/* D est l'ensemble des variables des distances de dépendance dans Sys;  */
/* I est l'ensemble des variables du système Sys n'appartenant pas à D.  */
/* La procédure retourne une (ou plusieurs) abstraction des dépendances  */
/* si une dépendance existe.                                             */

/* Effectuer le test de constante pour chaque équation du système Sys    */
if (TestConstante(Sys) = false)
        return(null)
elseif (TestGCD (Sys) = false)
        return(null)
elseif (PreTestLexicoPosit(Sys) = false)
        return(null)
elseif (TestConsist(Sys) = false)
        return(null)
/* Effectuer la projection de Sys(I, D) sur D. Le Sys(I,D) devient Sys(D). */
elseif (TestFaisabilite(Sys, I)) = false)
        return(null)
else SysD = Sys
if (TestFaisabilite(Sys, D)) = false)
        return(null)
elseif (TestLexicoPosit(SysD) = false)
        return(null)
/* Le calcul des profondeurs et du cône de dépendance.                   */
else AD = CalculApproxDep(SysD)
return (AD)
```

FIG. 2.5 - *Algorithme 2.2. Analyse de Dépendance (version 1)*

2.4.6 Améliorations

Améliorations de l'algorithme 2.2

Dans le cadre de cette thèse, nous avons apporté de nombreuses améliorations et modifications à l'algorithme d'analyse des dépendances, précédemment utilisé dans **PIPS**. Ces améliorations consistent essentiellement en :

- un découpage de l'algorithme en plusieurs tests et étapes permettant de supprimer les redondances,
- un réordonnancement des tests : le test du *GCD* a été extrait du test de *Consistance* pour être appliqué lors de la construction du système; le *Pre-Test de Lexico-Positivité* a été extrait du *Test de Lexico-Positivité* pour être appliqué avant le test de *Consistance*.
- une amélioration du test de *Faisabilité* pour que l'élimination des variables par propagation des équations soit effectuée avant celle utilisant l'algorithme de Fourier-Motzkin;
- l'implémentation du calcul du cône de dépendance (voir le chapitre 3).

D'autre part, des corrections significatives ont été effectuées afin de tenir compte des trois cas suivants : (1) présence de variables scalaires, dans la fonction d'accès aux éléments du tableau, qui ne sont pas des indices de boucle mais qui sont néanmoins modifiées dans le corps de boucles (2) présence d'un pas de boucle non numérique, (3) présence d'instructions non structurées et de tests IF dans le corps de boucles.

Nouvelle version de l'analyse de dépendance

Le test de dépendance de l'algorithme 2.2 effectue deux tests de dépendance pour chacune des paires de références $(S1, R1)$ et $(S2, R2)$. Il teste l'existence des deux dépendances possibles $dep1 : (S1, R1) \rightarrow (S2, R2)$ et $dep2 : (S2, R2) \rightarrow (S1, R1)$.

Sachant que :

- les résultats des tests : de *Constantes*, du *GCD*, de *Consistance* et de *Faisabilité* sont identiques pour les systèmes $Sys1(dep1)$ et $Sys2(dep2)$,
- et que les systèmes résultants de la projection de $Sys1(D)$ et $Sys2(D)$ sur D sont complémentaires : $Sys1(D) = Sys2(-D)$;

La première amélioration de la phase d'analyse des dépendances consiste à diminuer le nombre de dépendances testées en interprétant les résultats obtenus au cours du premier test pour le second. Nous avons donc modifié la procédure de calcul du graphe de dépendance afin de n'appliquer qu'une seule fois l'algorithme de test pour les deux références. Cette nouvelle version est présentée dans l'algorithme 2.3.

Dans cette nouvelle version, les tests *PreTestLexicoPosit(Sys)* et *TestLexicoPosit(SysD)* de l'algorithme 2.2 ont été remplacés respectivement par *PreTestAllEquals(Sys-R1R2)* et *TestAllEquals(Sys-R1R2)*. Ces deux derniers ne tiennent pas compte des dépendances *loop-independent* sur une même instruction (i.e. $S1 = S2$).

2.5 Expériences

La performance d'un test de dépendance dépend essentiellement de deux critères : sa *précision* et son *efficacité*. La précision du test est dépendante des informations (resultats des phases d'analyse sémantique et interprocédurale) que l'on introduit dans le système de dépendance et de la précision de l'algorithme de résolution du système utilisé.

Le test de dépendance de **PIPS** possède trois options correspondant aux trois systèmes de dépendances, proposés en section 2.4.4, qui ont des précisions différentes. L'algorithme de résolution du système de dépendance *Algorithme 2.2* ou *Algorithme 2.3* n'est pas un test exact et sa complexité est, théoriquement, exponentielle. Cependant, dans des cas particuliers, cet algorithme est exact (voir

```
procedure AnalyseDep(Sys-R1R2)
/* (R1,R2) est une paire des références à analyser;                          */
/* Sys-R1R2 est le système de dépendance de R1 à R2;                         */
/* D est l'ensemble des variables des distances dans Sys-R1R2;               */
/* I est l'ensemble des variables différentes que D dans Sys-R1R2.           */

if (TestConstante (Sys-R1R2) = false)
        return(null, null)
elseif (TestGCD (Sys-R1R2) = false)
        return(null, null)
elseif (PreTestAllEquals (Sys-R1R2) = false)
        return(null, null)
elseif (TestConsist (Sys-R1R2) = false)
        return(null, null)
/* Effectuer les projections de Sys-R1R2(I,D) sur D.                         */
elseif (TestFaisabilite (Sys-R1R2, I) = false)
        return(null, null)
else SysD-R1R2(D) = Sys-R1R2(D),    SysD-R2R1(D) = Sys-R1R2(-D)
if (TestFaisabilite (Sys-R1R2, D) = false)
        return(null, null)
elseif (TestAllEquals (Sys-R1R2) = false)
        return(null, null)
else {
        /* Calcul des profondeurs et du cône de dépendance pour la dépendance de R1 à R2. */
        if (LexicoPositif (SysD-R1R2)) AD-R1R2 = CalculApproxDep(SysD-R1R2)
        else AD-R1R2 = null
        /* Calcul des profondeurs et du cône de dépendance pour la dépendance de R2 à R1. */
        if (LexicoPositif (SysD-R2R1)) AD-R2R1 = CalculApproxDep(SysD-R2R1)
        else AD-R2R1 = null
}
return (AD-R1R2, AD-R2R1)
```

FIG. 2.6 - *Algorithme 2.3. Analyse de Dépendance (version 2)*

la section 2.3) et sa complexité moyenne est polynomiale si la taille des systèmes est petite.

Dans le but d'évaluer la performance de notre test de dépendance, nous avons effectué une série de mesures sur des programmes réels. La première évaluation expérimentale a été faite en Février 1992 avec le premier algorithme de test 2.2 [Yang92]. Cette évaluation a été effectuée sur le benchmark du PerfectClub qui contient 13 programmes scientifiques collectés par l'Université de l'Illinois en 1987. Cet expérience préliminaire a permis de soulever les problèmes qu'il était intéressant de résoudre pour l'obtention de bons résultats. Les trois problèmes principaux étaient les suivants : (1) le dernier test qui était effectué, le test de lexico-positivité avait un taux de succès de 35%; (2) les expériences ne pouvaient s'effectuer que sur les boucles structurées et ne contenant pas de tests IF; (3) le test de dépendance était appliqué deux fois pour chaque paire de références.

La deuxième évaluation a été faite en Mars 1993 sur le même benchmark avec la nouvelle version du test de dépendance, l'algorithme 2.3, où les trois problèmes précédents ont été résolus. Les expériences que nous avons effectuées consistaient à comparer : 1) la précision des trois systèmes de dépendance; 2) l'efficacité des 6 tests utilisés par l'algorithme; 3) l'exactitude de l'algorithme; 4) la complexité moyenne de l'algorithme et 5) le gain obtenu par l'algorithme 2.3 par rapport à l'algorithme 2.2.

2.5.1 Comparaison des systèmes de dépendance

L'un des points forts de **PIPS** est son analyse sémantique intra- et inter-procédurale. Comme il y a plusieurs analyses, il y a plusieurs sous-versions du test "sémantique". Des options permettent de prendre en compte les différents résultats qu'elles apportent.

La première étape de l'évaluation consistait à évaluer le nombre d'indépendances trouvées en utilisant les trois options du test de dépendance de **PIPS**.

Les résultats de cette expérience sont présentés par le tableau 2.1. Nous définissons l'*amélioration* apportée par le *test2* par rapport au *test1* de la manière suivante.

$$amelioration = \frac{\#independance_{test2} - \#independance_{test1}}{\#independance_{test1}}$$

Nous avons constaté que l'*amélioration* apportée par l'utilisation du *test complet*[6] par rapport au *test normal* est de 2.98% et que celle apportée par le *test normal* par rapport au *test simple* est de 3.49% . L'importance de l'utilisation de l'analyse sémantique (intraprocédurale) est donc très importante.

Remarquons qu'il est plus intéressant d'évaluer le nombre d'indépendances que le nombre de dépendances car lorsqu'un système est déclaré *indépendant*, on est sûr que les références sont indépendantes. Il est donc important de détecter au plus tôt ce type de système. Dans le cas des systèmes *dépendants* le test devra être effectué totalement.

Programme	#tests	test simple	test normal			test complet		
		#indep.	#indep.	diff.	amélio.	#indep.	diff.	amélio.
APS(adm)	1318	592	606	14	2.36%	709	103	17%
CS(spice)	5672	644	692	48	7.45%	693	1	0.15%
LG(qcd)	7733	6462	6565	103	1.59%	6613	48	0.73%
LW(mdg)	932	73	200	127	173.97%	200	0	0%
MT(track)	244	100	102	2	2%	109	7	6.86%
NA(bdna)	2058	338	339	1	0.30%	429	90	26.55%
OC(ocean)	805	75	93	18	24%	98	5	5.38%
SD(dyfesm)	1064	671	693	22	3.28%	726	33	4.76%
SM(mg3d)	968	128	128	0	0%	135	7	5.47%
SR(arc2d)	2881	2253	2256	3	0.13%	2310	54	2.39%
TF(flo52)	1675	1133	1201	68	6%	1205	4	0.33%
TI(trfd)	103	1	1	0	0%	1	0	0%
WS(spec77)	1927	294	333	39	13.27%	375	42	12.61%
Total	27380	12764	13209	445	3.49%	13603	394	2.98%

TAB. 2.1 - *Comparaison de trois systèmes sur le nombre d'indépendances détectées*

6. utilisant une analyse sémantique intraprocédurale

2.5.2 Performances des tests de dépendance

L'algorithme 2.3 est composé de 6 tests. L'ordre d'appel des ces tests est détaillé en section 2.4.6. Lors de la première évaluation de l'algorithme 2.2, les expériences avaient montré [Yang92] que le dernier test de lexico-positivité avait un fort taux de succès. C'était ce test qui conduisait dans de nombreux cas à la conclusion qu'il n'y avait pas de dépendance possible. Le test *PreTestLexicoPosit* (ou *PreTestAllEquals*) a donc été ajouté de manière à détecter cette *indépendance* plus tôt, et principalement dans les cas où le vecteur de distance est constant et lexico-négatif.

Les nouvelles mesures effectuées sur le nombre d'indépendances détectées par chaque test pour l'algorithme 2.3 (avec utilisation du *test complet*) sont détaillées dans le tableau 2.2. Les taux de succès pour chacun des tests sont détaillés dans le tableau 2.3. Nous constatons que l'appliquation de ce test pre-lexico-positif permet d'augmenter le nombre d'*indépendances* détectées avant l'application des tests plus complexes de 16%.

Nous observons aussi que, dans la majorité des cas (93.55%), les trois premiers tests simples, de complexité théorique polynômiale, permettent de conclure à *l'indépendance*.

Programme	#Tests	#indep.	const.	GCD	prelexi-pos.	consist.	faisab.	lexi-pos.
AP(adm)	1318	709	166	106	320	6	53	58
CS(spice)	5672	693	570	0	74	6	42	1
LG(qcd)	7733	6613	6441	0	21	24	79	48
LW(mdg)	932	200	27	0	16	71	56	30
MT(track)	244	109	31	0	69	0	3	6
NA(bdna)	2058	429	15	0	323	0	37	54
OC(ocean)	805	98	0	6	65	13	8	6
SD(dyfesm)	1064	726	533	0	138	0	43	12
SM(mg3d)	968	135	18	63	43	7	0	4
SR(arc2d)	2881	2310	1751	0	502	1	30	26
TF(flo52)	1675	1205	802	9	322	51	21	0
TI(trfd)	103	1	0	0	1	0	0	0
WS(spec77)	1927	375	18	0	274	20	30	33
Total	27380	13603	10372	184	2168	199	402	278
Taux		100%	76.2%	1.35%	16%	1.5%	3.0%	2.0%

TAB. 2.2 - *Nombre d'indépendances détectées à chaque test*

Programme	#Tests	#indep.	const.	GCD	prelexi-pos.	consist.	faisab.	lexi-pos.
AP(adm)	1318	709	23.4%	14.9%	45.1%	0.9%	7.5%	8.2%
CS(spice)	5672	693	82.2%	0%	10.7%	0.9%	6.1%	0.1%
LG(qcd)	7733	6613	97.4%	0%	0.3%	0.4%	1.2%	0.7%
LW(mdg)	932	200	13.5%	0%	8%	35.5%	28%	15%
MT(track)	244	109	28.4%	0%	63.3%	0%	2.8%	5.5%
NA(bdna)	2058	429	3.5%	0%	75.3%	0%	8.6%	12.6%
OC(ocean)	805	98	0%	6.1%	66.3%	13.3%	8.2%	6.1%
SD(dyfesm)	1064	726	73.4%	0%	19.0%	0%	5.9%	1.7%
SM(mg3d)	968	135	13.3%	46.7%	31.8%	5.2%	0%	3.0%
SR(arc2d)	2881	2310	75.8%	0%	21.7%	0.1%	1.3%	1.1%
TF(flo52)	1675	1205	66.6%	0.8%	26.7%	4.2%	1.7%	0%
TI(trfd)	103	1	0%	0%	100%	0%	0%	0%
WS(spec77)	1927	375	4.8%	0%	73.1%	5.3%	8%	8.8%
Total	27380	13603	76.2%	1.35%	16%	1.5%	3.0%	2.0%

TAB. 2.3 - *Taux de succès de chaque test*

2.5.3 Exactitude de l'algorithme

L'algorithme du test de dépendance de **PIPS** effectue une serie de tests approximatifs (pas toujours exacts en entiers). Quand le système de dépendance est faisable, on ne peut pas garantir l'existence d'une solution entière au système. Ces tests sont néanmoins conservatifs car dans le doute les références sont déclarées dépendantes. La perte de précision vient de l'algorithme de projection des variables qui n'est pas exact en entiers. L'algorithme élimine les variables en plusieurs étapes 1) par suppression des équations 2) par l'algorithme de Fourier-Motzkin. Cependant, cette projection reste dans certains cas exacte, lorsque :

- L'élimination de variable k dans l'équation

$$ak = A$$

  est exacte si et seulement si la valeur absolue de a vaut 1.

- L'élimination de Fourier-Motzkin est exact si une des trois conditions suffisantes proposées par C. Ancourt [Anco91] est vérifiée (voir la section 2.3).

Pour évaluer la précision de notre test de dépendance, nous avons mesuré le nombre de dépendances qui n'etaient pas exactes par rapport au nombre de tests

total. Le résultat de cette étude est illustrée par le tableau 2.4. L'exactitude de notre algorithme atteint 97.95% sur les expériences que nous avons faites.

D'après nos expériences, nous observons que la majorité des dépendances, qui ne sont pas nécessairement de vraies dépendances, sont provoquées par l'élimination (non exacte) d'une équation. Pour le programme MT(track), elles sont provoquées par l'élimination des variables au sein des inéquations par l'algorithme de Fourier-Motzkin. L'exactitude de notre algorithme peut donc être encore améliorée si on utilise un algorithme d'élimination des équations *exact* comme celui du GCD généralisé.

Il faut aussi remarquer que les conditions que nous avons utilisé pour tester l'exactitude de l'élimination d'une variable par Fourier-Motzkin sont des conditions *suffisantes* mais pas *nécessaires*. L'exactitude de l'algorithme de **PIPS** que nous avons obtenue est donc *approximative* mais néanmoins suffisante.

Programme	tests	tests exacts		tests-inexact
		independ.	depend.-exact	depend.-inexact
AP(adm)	1318	709	605	4
CS(spice)	5672	693	4979	0
LG(qcd)	7733	6613	1120	0
LW(mdg)	932	200	732	0
MT(track)	244	109	129	6
NA(bdna)	2058	429	1464	165
OC(ocean)	805	98	706	1
SD(dyfesm)	1064	726	338	0
SM(mg3d)	968	135	604	229
SR(arc2d)	2881	2310	571	0
TF(flo52)	1675	1205	470	0
TI(trfd)	103	1	102	0
WS(spec77)	1927	375	1396	156
		13610	13227	
Total	27380	26819		561
Taux		97.95%		2.05%

TAB. 2.4 - *Exactitude de l'algorithme*

2.5.4 Complexité moyenne de l'algorithme

Théoriquement, l'algorithme du test de dépendance de **PIPS** a une complexité exponentielle, dûe à l'emploi de l'algorithme d'élimination d'une variable de Fourier-Motzkin[7]. Cependant, dans la pratique, cet algorithme est souvent polynômial puisque des résultats expérimentaux montrent que la plupart des références aux tableaux dans les programmes scientifiques Fortran sont simples [SLY89]. Afin d'évaluer la complexité moyenne de l'algorithme du test de dépendance, nous commencons par calculer la complexité moyenne *pratique* de l'algorithme de Fourier-Motzkin. Nous détaillons cette évaluation dans cette section.

Complexité moyenne de l'algorithme de Fourier-Motzkin

Pour un système de n inéquations linéaires à m variables,

$$SysIneg : \sum_{j=1}^{m} a_{ij}x_j \ \geq \ b_i, \ i = (1, ..., n)$$

sous les hypothèses que le système d'inéquations est plein (i.e. chaque inéquation contraint toutes les variables) et est composé d'autant d'inéquations positives que négatives pour chacune des variables, alors la complexité de Fourier-Motzkin est donnée par la formule suivante [Duff74].

$$C_{FM} \ = \ O \ (\ 4 \ (\ n/4 \ )^{2^m} \ )$$

Pour un système peu dense dont le nombre d'inégalités contraignant chaque variable est inférieur à 4, la complexité réelle calculée est :

$$C_{FM} \ = \ C_{v_1} + C_{v_2} + ... + C_{v_m}$$

Puisque $C_{v_i} \leq O(1)$ $(1 \leq i \leq m)$,

$$C_{FM} \leq O(m)$$

ce résultat reste polynomial.

7. qui est exponentiel dans le pire des cas

Pour évaluer la complexité pratique de l'algorithme de Fourier-Motzkin pour le test de dépendance, nous évaluons la taille du système avant toute élimination par l'algorithme de Fourier-Motzkin et calculons le pourcentage de cas où la taille du système augmente après élimination. La table 2.5 illustre ces mesures. La première colonne relate le nombre d'éliminations de variable utilisant l'algorithme de Fourier-Motzkin; la deuxième colonne contient le nombre de cas où la taille du système augmente; n est le produit du nombre d'inégalités positives et négatives contraignant la variable éliminée.

Dans 99.73% des cas, n est inférieur à 4 et le pourcentage de projections faisant augmenter la taille du système reste inférieur à 0.3688%. La complexité réelle de l'algorithme de Fourier-Motzkin, dans le cadre d'un test de dépendance, reste donc en moyenne polynômiale et tout à fait acceptable.

Programme	#Elimi	#Sys.Aug.	n=0	n=1	n=2	n=3	n=4	n≥5
AP(adm)	4420	18	4064	115	39	3	185	14
CS(spice)	17886	2	9129	4331	14	0	4412	0
LG(qcd)	10971	7	7993	2257	2	0	719	0
LW(mdg)	6968	170	6142	111	0	0	576	139
MT(track)	612	5	485	31	2	0	82	12
NA(bdna)	10695	27	9478	713	48	4	447	5
OC(ocean)	5933	2	4957	538	4	0	434	0
SD(dyfesm)	1953	35	1665	110	0	0	174	4
SM(mg3d)	11610	0	11325	12	14	0	259	0
SR(arc2d)	3717	0	3179	262	0	0	276	0
TF(flo52)	1950	0	1673	75	0	0	202	0
TI(trfd)	1834	45	1589	38	40	0	103	64
WS(spec77)	9308	13	5235	2189	8	0	1876	0
#Elimi	87857	324	66914	10782	171	7	9745	238
Pourcentage	100%	0.3688%	76.16%	12.27%	0.20%	0.01%	11.09%	0.27%

#Elimi : Nombre d'éliminations de Fourier-Motzkin effectuées
#Sys.Aug : Nombre de systèmes dont la taille augmente après une projection
n : Produit des nombres d'inégalités positives et négatives

TAB. 2.5 - *Evaluation des tailles du système pendant l'élimination par Fourier-Motzkin*

Complexité moyenne de l'algorithme

Tenant compte de la complexité moyenne de l'algorithme de Fourier-Motzkin calculée précédemment, nous calculons maintenant la complexité moyenne de l'algorithme 2.3.

Supposons que le système de dépendance est composé de : n contraintes dont n_{eq} équations et m variables dont m_d variables de distance.

- les tests *TestConstante* et *TestGCD* sont effectués au moment de la construction de système. Au pire leur complexité est
  $C_{TestConstante,TestGCD} \approx O(n_{eq} \cdot m)$;
- *PreTestAllEquals(Sys-R1R2)* teste si la valeur prise par chacune des variables de distance est constante, en examinant les équations du système. Sa complexité est au plus $C_{PreTestAllEquals} \approx O(m_d \cdot n_{eq})$;
- *TestConsist(Sys-R1R2)* compare chaque contrainte du système avec toutes les autres. On a donc $C_{TestConsist} = O(n^2)$;
- les tests *TestFaisabilite(Sys-R1R2, I)* et *TestFaisabilite(SysD-R1R2,D)* éliminent au total n variables en utilisant l'algorithme de Fourier-Motzkin. D'après le résultat précédent, la complexité moyenne de ce test est
  $C_{TestFaisabilite} \approx O(m)$;
- *TestAllEquals(SysD-R1R2)* évalue les bornes minimale et maximale pour chaque variable de distance et utilise aussi Fourier-Motzkin, d'où :
  $C_{TestAllEquals} \approx O(m_d^2)$.

La complexité de l'algorithme 2.3 est, dans le pire des cas :

$$C_{algorithme2.3} \approx O(n^2 + m_d^2 + (m + m_d) \cdot n_{eq} + m)$$

qui est polynomiale.

2.5.5 Comparaison des deux versions du test de dépendance

Le fait que l'algorithme 2.2 analyse toute les dépendances référence à référence (ex. $R1 \rightarrow R2$) tandis que l'algorithme 2.3 analyse les dépendances globalement par paire de références (ex. $R1 \rightarrow R2$ et $R2 \rightarrow R1$) constitue la principale différence entre ces deux algorithmes. Le calcul du graphe de dépendances par l'algorithme 2.3 diminue considérablement le nombre de dépendances testées. Pour évaluer cette amélioration, nous avons mésuré le nombre de tests appliqués par ces deux algorithmes pour notre benchmark. D'après le tableau 2.6, l'utilisation de l'algorithme 2.3 diminue le nombre de tests appliqués de 46.06% (i.e. il est presque divisé par 2).

Programme	#Tests avec l'algorithme 2.2	#Tests avec l'algorithme 2.3
APS(adm)	2179	1318
CS(spice)	11056	5672
LG(qcd)	15232	7733
LW(mdg)	1768	932
MT(track)	380	244
NA(bdna)	3588	2058
OC(ocean)	1466	805
SD(dyfesm)	1805	1064
SM(mg3d)	1364	968
SR(arc2d)	5332	2881
TF(flo52)	3096	1675
TI(trfd)	174	103
WS(spec77)	3328	1927
Total	50768	27380
Réduction	0%	46.06%

TAB. 2.6 - *Comparaison des deux tests de dépendances sur le nombre total de tests effectués*

2.6 Comparaison de PIPS avec d'autres prototypes

La performance d'un paralléliseur dépend essentiellement des deux phases suivantes : la **détection du parallélisme** et l'**exploitation du parallélisme**.

Pour comparer les performances de différents paralléliseurs, on compare souvent le taux de *parallélisme* obtenu par chacun des paralléliseurs. Le taux de parallélisme d'un programme paralléle peut être évalué dynamiquement par le calcul du *speedup (i.e.* T_1/T_p[8]*)* obtenu sur une machine parallèle avec p processeurs, ou statiquement par l'évaluation du nombre de boucles parallèles du programme parallélisé. De nombreuses études portant sur ce type de comparaisons sont présentées dans [EiBl91], [ChPa91] et [LCD91]. Cependant, comme la performance du paralléliseur dépend à la fois de la précision du test de dépendance et de l'efficacité de la phase de parallélisation, ce type de comparaisons ne peut pas être exploité pour comprendre exactement la raison de l'obtention des différents résultats. Nous comparons donc ici uniquement les résultats obtenus par différents tests de dépendance.

La comparaison des résultats obtenus par différents tests de dépendance sur un même benchmark permettent de corriger et améliorer ces tests en réduisant leurs points faibles. Dans ce but, nous avons comparé nos résultats, sur le nombre d'indépendances détectées, avec les mesures obtenues par d'autres prototypes universitaires comme PFC [GKT91], SUIF [MHL91] et PARAFRASE [PePa91] sur un même benchmark, celui du Perfect Club.

2.6.1 Première comparaison

Les résultats de notre première comparaison, effectuée en 1992, sont illustrés par le tableau 2.7.

Nous avons constaté que ces mesures n'étaient pas directement comparables, ne serait-ce que parce que les nombres de paires de références aux tableaux testées sont différents. Cela s'explique principalement pour les raisons suivantes: (1) les

8. T_1 est le temps d'exécution sur un seul processeur, T_p est le temps d'exécution sur p processeurs

Perfect	#paires de références testées				#indépendances trouvées			
Program	PFC	SUIF	PIPS	CSRD	PFC	SUIF	PIPS	CSRD
adm	1328	1360	1444	18178	379	369	957	6615
arc2d	4969	2878	3678	23887	4132	1784	3195	21062
bdna	2056	2031	2460	10380	50	76	557	4982
dyfesm	1148	1066	631	5962	636	566	459	4038
flo52	1934	1644	2125	7274	1059	853	1844	6735
mdg	933	1045	222	7208	415	154	71	1712
mg3d	974	3833	-	30936	75	215	-	15023
ocean	468	3194	619	4150	11	73	144	1519
qcd	7948	7639	2356	142223	6731	6528	1424	77525
spc77	1718	1881	1694	11891	550	307	447	6253
spice	521	6155	474	30688	120	1166	84	3061
track	357	238	117	2743	100	34	91	1711
trfd	103	103	125	7278	0	0	1	3331

TAB. 2.7 - *Comparaison des mesures du test de dépendance de 4 prototypes (Juin 1992)*

systèmes de dépendances sont différents, (2) la définition de l'*indépendance* n'est pas la même pour chacun des prototypes. Nous détaillons maintenant le point commun et les différences entre les différents prototypes étudiés.

Point commun

Les tests de dépendances s'effectuent sur des paires de références à des tableaux référencées au sein d'au moins une boucle commune.

Différences

PFC

- un *seul* test pour toute paire de références à un tableau $(R1, R2)$ est utilisé pour détecter l'existence de dépendances possibles pour $R1 \delta R2$ et $R2 \delta R1$,
- les dépendances *loop independent*[9] entre deux références d'une même instruction ne sont pas comptées dans les dépendances.

9. internes à une boucle

SUIF

- un *seul* test pour toute paire de références à un tableau $(R1, R2)$ est utilisé pour détecter l'existence de dépendances possibles pour $R1\delta R2$ et $R2\delta R1$,
- les références de type *complexes* sont divisées en parties réelle et imaginaire. Pour chaque référence complexe, deux tests sont appliqués.
- les dépendances *loop independent* entre deux références d'une même instruction sont comptées parmi les dépendances,
- un préprocesseur applique des transformations de programme telles que : détection des réductions, élimination de codes morts, élimination de variables inductives, etc.

PIPS

- seules les dépendances contenues dans les boucles, qui ne contiennent pas de structure de contrôle comme des IFs et GOTOs, sont testées,
- deux tests sont effectués pour toute paire de références $(R1, R2)$ à un tableau lors de la détection de l'existence de dépendances possibles pour $R1\delta R2$ et $R2\delta R1$,
- les dépendances *loop independent* entre deux références d'une même instruction ne sont pas comptées dans les dépendances,

CSRD

- le nombre de tests effectués correspond au nombre de tests intervenant pour chaque paire de références aux tableaux pour chaque *ddv*.

Chaque prototype ayant son propre environnement de test de dépendance, la comparaison de ces tests reste difficile. Toutefois, nous avons modifié et amélioré le test de de dépendance de Pips afin d'effectuer de meilleures comparaisons.

2.6.2 Deuxième comparaison

Nous avons modifié l'algorithme du test de dépendance de Pips pour qu'il teste les dépendances pour tout type de boucles, y compris celles contenant des IFs et GOTOs. Nous avons amélioré l'algorithme de manière à ce qu'un seul test soit effectué pour toute paire de références $(R1, R2)$ à un tableau lors de la détection des dépendances possibles $R1\delta R2$ et $R2\delta R1$. Pour être plus proche des résultats obtenus par les autres prototypes, une variable supplémentaire `LID` a été ajoutée pour compter les dépendances *loop-independent* au sein d'une même instruction. Le tableau 2.8, illustre les nouvelles mesures. Dans la seconde partie du tableau, les chiffres indiqués pour Pips donnent à la fois le nombre d'indépendances total et celui après suppression des `LID` dépendances *loop-independent*.

Les mesures du test de dépendance du CSRD ne sont pas présentées car elles sont incomparables avec celles des autres prototypes.

Perfect	#paires de références testées			#indépendances trouvées			
Program	PFC	SUIF	PIPS	PFC	PIPS	SUIF	PIPS
adm	1328	1360	1318	379	709	369	331
arc2d	4969	2878	2881	4132	2310	1784	1782
bdna	2056	2031	2058	50	429	76	52
dyfesm	1148	1066	1064	636	726	566	576
flo52	1934	1644	1675	1059	1205	853	883
mdg	933	1045	932	415	200	154	154
mg3d	974	3833	968	75	135	215	88
ocean	468	3194	805	11	98	73	27
qcd	7948	7639	7733	6731	6613	6528	6544
spc77	1718	1881	1927	550	375	307	68
spice	521	6155	5672	120	693	1166	618
track	357	238	244	100	109	34	34
trfd	103	103	103	0	1	0	0

TAB. 2.8 - *Comparaison des mesures du test de dépendance de 3 prototypes (Juin 1993)*

Les nouvelles mesures, illustrées dans le tableau 2.8, permettent une meilleure comparaison des résultats obtenus par **PIPS**. Excepté pour le programme **ocean**, le nombre de paires de références testées par **PIPS** est compatible avec celui des

autres prototypes.

Pour poursuivre les comparaisons, il faudrait utiliser une même structure de données pour le graphe de dépendances (ex. PSERVE de Rice) ou exécuter tous les systèmes sur le même code transformé. Cette direction n'a pas être poursuivie faute de coopération de la part de nos partenaires.

2.7 Conclusion

Nous avons présenté dans ce chapitre, le test de dépendance, une des phases essentielles de la parallélisation. Le test de dépendance est composé de deux tâches principales : la construction du système de dépendance, linéaire en général, et la résolution en nombres entiers de ce système. De nombreurs chercheurs ont consacré leur recherche à cette deuxième tâche.

Nous avons présenté, dans la première partie de ce chapitre, différentes méthodes d'approximation de résolution des systèmes linéaires de dépendance et certaines conditions pour lesquelles ces tests approximatifs conduisent toutefois à un résultat exact. Nous avons détaillé, dans la deuxième partie, l'algorithme du test de dépendance d'un paralléliseur **PIPS** ainsi que les améliorations que nous y avons apporté dans le cadre de cette thèse.

Le test de dépendance de **PIPS** contient trois types de systèmes de dépendance et utilise un algorithme approximatif (résolution rationnelle) avec une complexité théorique exponentielle.

Nous avons présenté, à la fin de ce chapitre, les résultats de l'évaluation expérimentale des performances de notre algorithme selon les critères :

1. précision des trois systèmes de dépendance;
2. efficacité des 6 tests utilisés par l'algorithme;
3. exactitude;
4. complexité moyenne.

D'après nos expériences, nous avons constaté que :

1. l'analyse sémantique a un impact important sur le test de dépendance;

2. dans 93.55% des cas, les tests simples (de complexité polynômiale) suffisent pour détecter que le système traduit une indépendance;

3. un algorithme approximatif peut être suffisamment précis dans la pratique (notre algorithme permet l'obtention d'un résultat exact dans 97.95% des cas);

4. la taille des systèmes à résoudre, lors de l'élimination par Fourier-Motzkin, a dans 99.73% des cas un nombre de contraintes inférieur à 4;

5. la complexité moyenne de l'algorithme de Fourier-Motzkin est, dans la pratique, polynômiale;

6. la complexité moyenne du test de dépendance de **PIPS** est, dans la pratique, polynômiale.

Enfin, nous avons observé que de nombreuses solutions ont déjà été proposées pour résoudre un système linéaire dans le cadre d'un test de dépendance et que ces solutions apportent, dans la majorité des cas, de bons résultats. La perte de précision de certains tests vient principalement du manque d'information contenu dans les systèmes de dépendances. Ce manque d'information est dû parfois au fait que les hypothèses de linéarité ne sont pas respectées. Pour améliorer la précision du test dépendance, il faudrait utiliser des techniques non-linéaires [LiTh88] [Duma92] et proposer des extensions interactives faisant appel à l'utilisateur et dynamiques (analyse à *l'exécution*) au paralléliseur [Poly88] [Rose90] [SMC91] [LuCh91].

Chapitre 3

Abstraction des dépendances

Rappelons que la tâche du test de dépendance est de déterminer l'existence possible de collision lors de l'accès à une variable utilisée par deux instructions `S1` et `S2`. Lorsqu'un conflit possible existe, on dit qu'il y a une *dépendance* entre `S1` et `S2`. Cette dépendance est exprimée par un arc liant `S1` et `S2` dans *le graphe de dépendance*. Elle ne signifie pas forcément que toutes les instances de deux instructions `S1` et `S2` sont dépendantes, et de nombreuses techniques de transformation sont utilisées pour permettre d'exploiter le parallélisme implicite résiduel ou pour pouvoir l'exploiter au mieux sur une machine cible particulière.

Une transformation peut être appliquée au programme si toutes les contraintes sur les instances des instructions du programme sont respectées. Une représentation exacte ou approximative caractérisant ces contraintes est donc nécessaire.

De nombreuses abstractions ont été proposées dans la littérature : les itérations de dépendance, les vecteurs de distance de dépendance, le cône de dépendance, les vecteurs de direction de dépendance et les profondeurs de dépendance. Parmi elles, les vecteurs de direction de dépendance (DDV) sont largement utilisés. Cette approximation fournit l'information *minimale* nécessaire pour autoriser des transformations comme : l'échange de boucles et l'inversion de boucle. Par contre, elle n'est pas assez précise pour les transformations unimodulaires complexes et le partitionnement des boucles où l'information *minimale* nécessaire est le cône de dépendance.

Dans ce chapitre, nous détaillons les différents types d'abstraction des dépendances : *les itérations de dépendance, le vecteur de distance de dépendance, le*

polyèdre de dépendance, le cône de dépendance, le vecteur de direction de dépendance et la profondeur d'une dépendance. Ensuite nous comparons leur précision en fonction du nombre d'itérations dépendantes qu'elles traduisent (section 3.6). L'algorithme de calcul du polyèdre (et du cône) de dépendance est détaillé (section 3.3.3). Cet algorithme a été implanté dans le paralléliseur **PIPS**.

Rappelons quelques notations :

Notations :

- N : l'ensemble des entiers naturels;
- Z^n : l'ensemble n-uplets d'entiers;
- I^n : le domaine d'itérations;
- $\vec{i}^n$: une itération dans I^n;
- $\vec{i} \ll \vec{i'}$: $\vec{i}$ est plus petit que $\vec{i'}$ pour l'ordre lexicographique;
- $S1(\vec{i}) \prec S2(\vec{i'})$[1] : $S1(\vec{i})$ est exécutée avant $S2(\vec{i'})$;
- $\vec{i} \prec_\delta \vec{i'}$: $\vec{i}$ est exécutée avant $\vec{i'}$, les contraintes de dépendances sur les itérations sont conservées;
- $S1\ \delta^*_{AD}\ S2$: $S1$ dépend de $S2$, l'abstraction AD est utilisée pour représenter cette dépendance;
- $AD1 \supset AD2$: l'abstraction $AD1$ est plus précise que l'abstraction $AD2$

Nous supposons qu'il existe une dépendance entre deux instructions $S1$ et $S2$ au sein de `n` boucles englobantes. Nous détaillons maintenant les 6 différentes abstractions des dépendances qui permettent de caractériser les relations existant entre les deux itérations dépendantes $S1(\vec{i})$ et $S2(\vec{i'})$.

1. peut être traduit par le prédicat : $\vec{i} \ll \vec{i'}$ *ou* $(\vec{i} = \vec{i'} \wedge S1$ *est avant* $S2$ *textuellement*).

3.1 Itérations de Dépendance (DI)

Cette abstraction des dépendances, notée DI, consiste en une énumération exhaustive des paires d'itérations $(\vec{i},\ \vec{i'})$ de $S1(\vec{i})$ et $S2(\vec{i'})$ dépendantes.

Définition 3.1 $\quad DI = \{\ (\vec{i},\ \vec{i'}) \mid S1(\vec{i})\ \delta^*\ S2(\vec{i'})\ \}$

A chaque élément $(\vec{i},\ \vec{i'})$ de DI correspond une solution entière du système linéaire caractérisant les dépendances entre $S1(\vec{i})$ et $S2(\vec{i'})$. Aucune approximation n'est insérée dans DI, elle est très précise. Cette énumération peut être représentée sous plusieurs formes. Nous citons ici les trois exemples les plus représentatifs :

1. la liste exhaustive des couples des valeurs de toutes les itérations dépendantes,

2. l'introduction de paramètres entiers appropriés $\vec{t} = (t_1, t_2, ..., t_m)$ $(m < 2*n)$ peut permettre d'exprimer $\vec{i}$ et $\vec{i'}$ en fonction de $\vec{t}$. La représentation de DI devient
   $DI = \{\ (\vec{f}\,(\vec{t}),\ \vec{f'}\,(\vec{t})) \mid \vec{t} \in T^m\}$ où T est l'espace de $\vec{t}$ possédant des bornes connues.

3. lorsque $\vec{i'}$ peut s'exprimer sous la forme d'une fonction $\vec{f}$ de $\vec{i}$. La représentation de DI correspondante est
   $DI = \{\ (\vec{i},\ \vec{f}\,(\vec{i})) \mid \vec{i}\ \in I_s^n,\ I_s^n \subseteq I^n\}$

Prenons l'exemple suivant :

```
      DO I = 1, n
S1:      T1(I) = 4*I
         DO J = 1, n
S2:           T2(I,J) = T1(J)*T2(I,J)
         ENDDO
      ENDDO
```

FIG. 3.1 - *Exemple 3.1*

Considérons le système de contraintes caractérisant la dépendance *dep* entre $S1$ et $S2$ due aux deux références à `T1` :

$$\begin{cases} i = j \\ 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq i' \leq n \end{cases}$$

Ce système admet des solutions entières, `S1` dépend donc de `S2`. La représentation de cette dépendance $S1\ \delta^*_{DI}\ S2$ est la suivante :

```
DI (dep) = {  (1,1), (2,2), ..., (n-1,n-1), (n,n),
              (1,2), (2,3), ..., (n-1,n),
              (1,3), ..., (n-2,n),
              ...
              (1,n-1), (2,n),
              (1,n) }
```

Nous savons que pour calculer DI un algorithme exact de recherche des solutions entières dans un système linéaire est nécessaire. La représentation numérique (la première forme) de DI n'est pas acceptable en pratique, parce qu'elle nécessite trop de place mémoire. De plus, elle ne permet pas de représenter une dépendance ayant un nombre infini de paires d'itérations dépendantes.

L'extension du GCD généralisé [Bane88] a permis d'introduire la deuxième représentation de DI. L'espace des paramètres T doit toutefois être calculé exactement (voir la section 2.2.5).

L'algorithme de test de dépendance proposé par Feautier [Feau89], basé sur l'algorithme PIP [Feau88], permet de calculer les dépendances de flot de données (*data-flow dependence*) et de les représenter sous la troisième forme.

DI est l'abstraction des dépendances la plus précise, mais les algorithmes exacts permettant de la calculer sont tous très coûteux. De plus, la plupart des transformations n'ont pas besoin d'une information d'une telle précision pour savoir si on peut les effectuer. Pour ces raisons, des abstractions des dépendances approximant l'ensemble des dépendances exactes, telles que vecteur de distance de dépendance et ses dérivés, ont été proposées. Nous les présentons dans la suite de ce chapitre.

3.2 Vecteurs de Distance de dépendance (D)

Reprenons l'abstraction DI de l'exemple 3.1. Il y a des paires d'itérations $\vec{i}$, $\vec{i'}$ ($\vec{i} \prec_\delta \vec{i'}$) équi-distantes de $\vec{i'}$-$\vec{i}$. L'utilisation d'un vecteur de distance $\vec{d}$ permet de représenter uniformement toutes les dépendances entre les itérations $\vec{i}$ et $(\vec{i} + \vec{d})$. Elle reflète explicitement le pas de dépendance entre les itérations. Elle ne tient pas compte de l'origine des dépendances.

Représenter une dépendance par D revient à énumérer tous les vecteurs de distance possibles de cette dépendance. Voici une définition de D plus formelle.

Définition 3.2 $\quad D = \{ \vec{d} \mid \exists(\vec{i},\ \vec{i'}) \in DI,\ \vec{d} = \vec{i'} - \vec{i}\}$

Reprenons l'exemple 3.1, et représentons la dépendance sur $T1$ entre `S1` et `S2` par D :

```
D(dep) = { 0, 1, 2, 3, ..., n-1 }
```

Nous constatons que D représente les mêmes dépendances avec moins d'éléments que DI. Le calcul de D pour une dépendance nécessite aussi un test de

dépendance exact. De plus, D est éventuellement infinie. La complexité des tests exacts est, dans le cas général, exponentielle exceptée lorsque la distance de dépendance est constante. On utilise donc cette abstraction essentiellement lorsque les **dépendances** ont une distance constante, on dit qu'elles sont **uniformes**. Les dépendances uniformes de vecteur de distance de dépendance $\vec{d}$ représentent l'ensemble des itérations dépendantes : $DI = \{(\vec{i},\ \vec{i'}) \mid \vec{i'} = \vec{i} + \vec{d}\}$.

Dans la section suivante, nous présenterons deux dérivés de D : DC et DP qui sont basées sur des polyèdres convexes. Elles permettent de représenter de manière finie un ensemble infini de dépendances.

3.3 Polyèdre de Dépendance (DP) et Cône de Dépendance (DC)

Le **polyèdre de dépendance** (noté DP) [IrTr88b] approxime l'ensemble D. C'est l'ensemble des éléments qui sont **combinaison convexe** des vecteurs de D. Il est représenté par l'ensemble des points entiers de l'enveloppe convexe de D ou par une union de polyèdres convexes, lorsque D est un ensemble infini et que son enveloppe convexe contient des éléments autres que les combinaisons convexes des vecteurs de D. Le **cône de dépendance** (noté DC) est un cône convexe composé des vecteurs qui sont **combinaison linéaire positive** des points de D.

3.3.1 Utilisation de la théorie des polyèdres convexes

Afin de mieux caractériser DP et DC, nous rappelons quelques définitions et précisions sur les polyèdres convexes.

1. Un **polyèdre convexe** $P \in R^n$ est un ensemble de points qui satisfont un nombre fini d'inéquations linéaires.

2. Un **polyèdre** est **entier** si tous ses sommets sont entiers [Schr86]. D'après leur définition, DP et DC sont des polyèdres entiers.

3. Un polyèdre convexe entier peut être caractérisé soit par un système de contraintes linéaires diophantines, soit par un système générateur [Schr86].

   système de contraintes linéaires (noté SC) Un système de contraintes linéaires est composé d'un ensemble d'inéquations (une équation est considérée comme deux inéquations opposées). Un polyèdre `P` est un ensemble de points entiers satisfaisant un système d'inégalités.

   $P_{sc} = \{ \vec{x} \in Z^n \mid A\vec{x} \leq b \}$ où A est une matrice et b un vecteur.

   système générateur (noté SG) Un système générateur comprend trois ensembles finis : sommets $\{\vec{s_i}\}$, rayons $\{\vec{r_j}\}$, droites $\{\vec{d_k}\}$. En utilisant le système générateur, un polyèdre `P` peut être formulé par :

   $$P_{sg} = \{\vec{x} \in Z^n : \vec{x} = \sum_i \lambda_i \vec{s_i} + \sum_j \mu_j \vec{r_j} + \sum_k \nu_k \vec{d_k} \ , \ \lambda_i \geq 0 \ , \ \mu_j \geq 0 \ , \ \sum_i \lambda_i = 1\}$$

   Un format plus simple va être utilisé ultérieurement :
   $P_{sg} = (S, R, D)$, où $S = \{\vec{s_i}\}$, $R = \{\vec{r_j}\}$, $D = \{\vec{d_k}\}$

   Des algorithmes qui permettent de passer d'une représentation à l'autre ont été proposés dans la thèse d'Halbwachs [Halb79]. Un autre algorithme de conversion d'un système linéaire en système générateur a été proposé par Chernikova [Cher68].

4. Un polyèdre est **lexico-positif** si tous les points contenus dans ce polyèdre sont lexico-positifs. D'après leur définition, DP et DC sont des polyèdres lexico-positifs.

   Proposition 3.1 *Un polyèdre $P_{sg} = (S, R, D)$ est lexico-positif si et seulement si les conditions suivantes sont vérifiées :*

   - $\forall \ \vec{s_i} \in S, \quad \vec{s_i} \gg 0$
   - *si* $\exists \vec{r_j} \ll 0$ *, alors* $\forall \ \vec{s_i} \in S, \quad Level(\vec{r_j}) > Level(\vec{s_i})$
     où la fonction $Level(\vec{v})$ *est égale à* k *si les* $(k-1)$ *premières composantes de* $\vec{v}$ *sont nulles.*[2]

2. par exemple : $Level(0,0,1) = Level(0,0,-1) = 3, Level(0,0,0) = 4$

– *si* $D \neq \emptyset$, *alors* $\forall\ \vec{d_k}, \forall\ \vec{s_i},\quad Level(\vec{d_k}) > Level(\vec{s_i})$

Preuve :

– Montrons que si $P_{sg} = (S,\ R,\ D)$ est un polyèdre lexico-positif alors les trois conditions données ci-dessus sont vérifiées. Si $P_{sg} = (S,\ R,\ D)$ est un polyèdre lexico-positif alors tout point $\vec{v}$ de P_{sg} vérifie :

  – $\vec{v} = \sum_i \lambda_i \vec{s_i} + \sum_j \mu_j \vec{r_j} + \sum_k \nu_k \vec{d_k}\ , \sum_i \lambda_i = 1,\ \mu_j \geq 0$

  – $\vec{v} \gg 0$.

  (a) Montrons que tout sommet de P_{sg} vérifie la première condition :
  soit $s_i \in P_{sg}$ alors $\exists \lambda_i = 1, \vec{v} = \lambda_i \vec{s_i} \in P_{sg}$.
  Comme $\vec{v} \gg 0 \Longrightarrow \vec{s_i} \gg 0$

  (b) Montrons que tout rayon de P_{sg} vérifie la seconde condition. Montrons que
  si $\exists \vec{r_j} \ll 0\ et\ \exists\ \vec{s_i} \in S\ tel\ que\ \ Level(\vec{r_j}) \leq Level(\vec{s_i})$
  alors P_{sg} n'est pas lexico-positif.

  – Si $\exists \vec{r_j} \ll 0\ et\ \exists\ \vec{s_i} \in S\ tel\ que\ Level(\vec{r_j}) < Level(\vec{s_i})$, alors le point $\vec{v'} = \vec{s_i} + \vec{r_j}$ appartient à P_{sg}. Comme $Level(\vec{r_j}) < Level(\vec{s_i})$ nous avons $\vec{v'} \ll 0 \implies P_{sg}$ n'est pas lexico-positif.

  – Si $\exists \vec{r_j} \ll 0\ et\ \exists\ \vec{s_i} \in S\ tel\ que\ Level(\vec{r_j}) = Level(\vec{s_i}) = l$, alors le point $\vec{v'} = \vec{s_i} + \frac{1}{|r_{j_l}|}\vec{r_j} = (0, 0, ..., -1, *, ..., *)$ appartient à P_{sg}. Comme $\vec{v'} \ll 0 \implies P_{sg}$ n'est pas lexico-positif.

  (c) Montrons que toute droite de P_{sg} vérifie la troisième condition. Montrons que
  si $\exists \vec{d_k} = (d_{k_1}, ..., d_{k_n}),\ et\ \exists\ \vec{s_i} = (s_{i_1}, ..., s_{i_n})$
  $tel\ que\ \ Level(\vec{d_k}) = l\ et\ Level(\vec{d_k}) \leq Level(\vec{s_i})$, alors

P_{sg} n'est pas lexico-positif.

- Si $\exists \vec{d_k} = (d_{k_1}, ..., d_{k_n})$, *et* $\exists\ \vec{s_i} = (s_{i_1}, ..., s_{i_n})$ *tel que* $Level(\vec{d_k}) = l$ *et* $Level(\vec{d_k}) < Level(\vec{s_i})$, alors le point

$$\vec{v'} = \begin{cases} \vec{s_i} + \vec{d_k} & si \quad d_{k_l} < 0, \\ \vec{s_i} - \vec{d_k} & si \quad d_{k_l} > 0, \end{cases}$$

appartient à P_{sg}. Comme $Level(\vec{d_k}) < Level(\vec{s_i})$, nous avons $\vec{v'} \ll 0 \implies P_{sg}$ n'est pas lexico-positif.

- Si $Level(\vec{d_k}) = Level(\vec{s_i}) = l$, le point

$$\vec{v'} = \begin{cases} \vec{s_i} + \frac{1}{|d_{k_l}|}\vec{d_k} & si \quad d_{k_l} < 0, \\ \vec{s_i} - \frac{1}{|d_{k_l}|}\vec{d_k} & si \quad d_{k_l} > 0, \end{cases} = (0, 0, ..., -1, *, ..., *)$$

appartient à P_{sg}. Comme $\vec{v'} \ll 0 \implies P_{sg}$ n'est pas lexico-positif.

- Montrons que si les trois conditions de la proposition sont vérifiées alors le polyèdre est lexico-positif. Soit $P_{sg} = (S,\ R,\ D)$ un polyèdre et $\vec{s_i} \in S,\ \vec{r_j} \in R,\ \vec{d_k} \in D$ un sommet, un rayon et une droite vérifiant les trois conditions de la proposition. Tout point de P_{sg} s'écrit

$$\vec{v} = \sum_i \lambda_i \vec{s_i} + \sum_j \mu_j \vec{r_j} + \sum_k \nu_k \vec{d_k}\ ,\ \lambda_i \geq 0\ ,\ \sum_i \lambda_i = 1,\ \mu_j \geq 0\ ,$$

ou encore

$$\vec{v} = \sum_i \lambda_i \vec{s_i} + \sum_j \mu_j \vec{r_j} + \sum_{j'} \mu_{j'} \vec{r_{j'}} + \sum_k \nu_k \vec{d_k}$$

avec

$$\lambda_i \geq 0\ ,\ \sum_i \lambda_i = 1,\ \mu_j \geq 0\ ,\ \mu_{j'} \geq 0\ , \vec{s_i} \gg 0,\ \vec{r_j} \gg 0\ et\ \vec{r_{j'}} \ll 0$$

Comme pour tout rayon $\vec{r_{j'}} \ll 0$ et toute droite $\vec{d_k}$ nous avons $Level(\vec{r_{j'}}) > Level(\vec{s_i})\quad et\quad Level(\vec{d_k}) > Level(\vec{s_i})$, nous en déduisons que $\vec{v} \gg 0$.

Puisque tous les vecteurs composant $\vec{v}$ sont lexico-positifs, P_{sg} est un polyèdre lexico-positif. $\square$

5. L'ensemble des **combinaisons convexes** des points d'un ensemble X est égal à $\sum_{i=1}^{p} \lambda_i x_i$, où $x_i \in X$ et les λi sont des réels compris entre 0 et 1 [Schr86].

6. Le plus petit polyèdre convexe qui couvre toutes les combinaisons convexes d'un ensemble de points X est appelé l'**enveloppe convexe** de X (noté $env(X)$).

7. Soit $P_1 = (S_1,\ R_1,\ D_1)$ et $P_2 = (S_2,\ R_2,\ D_2)$ deux polyèdres, alors

$$env(P_1 \cup P_2) \ = \ (S_1 \cup S_2,\ R_1 \cup R_2,\ D_1 \cup D_2)$$

8. Soit $env(X)$ l' enveloppe convexe d'un ensemble de points de X et $conv(X)$ l'ensemble des combinaisons convexes des points de X. Les deux relations suivantes sont vérifiées:

   - $env(X) \ = \ conv(X)$ si X est un ensemble fini ou $X \ = \ env(X)$;
   - $env(X) \ \supseteq \ conv(X)$ si X est un ensemble infini. On a égalité lorsque $conv(X)$ forme un polyèdre (i.e. un ensemble fermé).

Proposition 3.2 *Soit $P_1 = (S_1,\ R_1)$ et $P_2 = (S_2,\ R_2)$ deux polyèdres pour lesquels chacune des droites est représentée par deux rayons opposés dans $R1$ (respectivement $R2$) pour $P1$ (respectivement $P2$). L'égalité $env(P_1 \cup P_2) \ = \ conv(P_1 \cup P_2)$ est vérifiée pour chacune des conditions suivantes :*

(a) L'un des deux polyèdres est inclus dans l'autre :

$$(P1 \ \supseteq \ P2) \ \vee \ (P2 \ \supseteq \ P1)$$

(b) Les polyèdres ne sont constitués que de sommets :

$$(R_1 = \emptyset) \ \wedge \ (R_2 = \emptyset)$$

(c) Les ensembles de rayons et droites des deux polyèdres sont identiques :

$$R_1 = R_2$$

Preuve :

(a) Supposons que $P_1 \supseteq P_2$, alors $P_1 \cup P_2 = P_1$.
Comme $P1$ est un polyèdre convexe, $env(P_1) = conv(P_1)$.
Nous déduisons que
$env(P_1 \cup P_2) = env(P_1) = conv(P_1) = conv(P_1 \cup P_2)$.

(b) Supposons que $(R_1 = \emptyset) \wedge (R_2 = \emptyset)$. Alors,
$env(P_1 \cup P_2) = (S_1 \cup S_2, \emptyset) = \{\vec{v} = \sum \lambda_i^1 s_i^1 + \sum \lambda_i^2 s_i^2 \mid \sum \lambda_i^1 + \sum \lambda_i^2 = 1, s_i^1 \in S_1, s_i^2 \in S_2\}$
D'après la définition de l'ensemble des combinaisons convexes d'un ensemble,
$conv(P_1 \cup P_2) = \{\vec{v} = \sum \lambda_i^1 \vec{v_1} + \sum \lambda_i^2 \vec{v_2} \mid \sum \lambda_i^1 + \sum \lambda_i^2 = 1, \vec{v_1} \in P_1, \vec{v_2} \in P_2\}$.
Puisque P_1 et P_2 ne sont composés que de sommets, $S_1 = \{ \vec{v_1} \}$ et $S_2 = \{ \vec{v_2} \}$, il en résulte que $env(P_1 \cup P_2) = conv(P_1 \cup P_2)$.

(c) Supposons que $R_1 = R_2 = R$. Alors,
$env(P_1 \cup P_2) = (S_1 \cup S_2, R) = \{\vec{v} = \sum \lambda_i^1 s_i^1 + \sum \lambda_i^2 s_i^2 + \sum \mu_j r_j \mid \lambda_i^1 \geq 0,\ \lambda_i^2 \geq 0, \sum \lambda_i^1 + \sum \lambda_i^2 = 1, \mu_j \geq 0\}$
Comme $\sum \lambda_i^1 + \sum \lambda_i^2 = 1$, il existe au moins un λ_i non nul. Sans perdre de généralité, nous prenons $\lambda_k^2 \neq 0$. Nous pouvons traduire $\vec{v}$ sous la forme suivante: $\vec{v} = \sum \lambda_i^1 s_i^1 + \sum_{i \neq k} \lambda_i^2 s_i^2 + \lambda_k^2 (s_k^2 + \sum \frac{\mu_j}{\lambda_k^2} r_j)$
Comme $s_k^2 \in S_2$, $\frac{\mu_j}{\lambda_k^2} \geq 0$ et $r_j \in R_2$, le vecteur $v^2 = s_k^2 + \sum \frac{\mu_j}{\lambda_k^2} r_j$ est un vecteur de P_2.
Posons $\vec{v} = \sum \lambda_i^1 s_i^1 + \sum_{i \neq k} \lambda_i^2 s_i^2 + \lambda_k^2 v^2$. Puisque s_i^1, s_i^2 et v^2 appartiennent à$P_1 \cup P_2$, et que $\sum \lambda_i^1 + \sum \lambda_i^2 = 1$, alors le vecteur $\vec{v}$ appartient aussi à $conv(P_1 \cup P_2)$.
Comme tout vecteur $\vec{v}$ de $env(P_1 \cup P_2)$ est inclus dans $conv(P_1 \cup P_2)$, la relation $env(P_1 \cup P_2) \subseteq conv(P_1 \cup P_2)$ est vérifiée. Comme par définition, $conv(P_1 \cup P_2) \subseteq env(P_1 \cup P_2)$, nous en déduisons que $env(P_1 \cup P_2) = conv(P_1 \cup P_2)$.

Proposition 3.3 *Soit P_1 et P_2 deux polyèdres lexico-positifs. Si $\neg(env(P_1 \cup P_2) \gg 0)$ alors $env(P_1 \cup P_2) \supset conv(P_1 \cup P_2)$.*

Preuve :

Comme $P_1 \gg 0 \ \wedge \ P_2 \gg 0$, par définition de $conv(P_1 \cup P_2)$,
$\Longrightarrow \ conv(P_1 \cup P_2) \gg 0$;
Par définition, $conv(P_1 \cup P_2) \subseteq \ env(P_1 \cup P_2)$
Nous en déduisons donc que
si $\neg(env(P_1 \cup P_2) \gg 0)$ alors $env(P_1 \cup P_2) \supset conv(P_1 \cup P_2)$. $\square$

3.3.2 Définitions

Nous définissons dans cette sous-section DP et DC.

Définition 3.3

$$DP = \{ \ \vec{v} \ \in Z^n : \exists \vec{d_i} \in D \ , \ \exists \lambda_i \geq 0 \ \ tel\ que \ \ \vec{v} = \sum_{1}^{k} \lambda_i \vec{d_i} \ \bigwedge \ \sum_{i=1}^{k} \lambda_i = 1 \}$$

L'abstraction des dépendances DP est une approximation de D qui conserve les informations "utiles" permettant d'effectuer certaines transformations comme les réordonnancements de boucles (ordonnancement linéaire) [Irig88a] (voir le chapitre suivant).

Le plus grand ensemble approximant les vecteurs de l'ensemble D et qui conserve les informations utiles aux réordonnancements de boucles est l'enveloppe convexe de la fermeture transitive de D . Il correspond à un cône convexe composé des vecteurs qui sont combinaison linéaire positive des points de D. Ce cône est le **cône de dépendance** (noté DC) [IrTr88b]. DP est inclus dans DC. Nous donnons la définition de DC :

Définition 3.4

$$DC = \{ \ \vec{v} \ \in Z^n \ | \ \exists \vec{d_i} \in D, \ \exists \lambda_i \geq 0, \ \sum_{i=1}^{k} \lambda_i \geq 1 \ \ tel\ que \ \ \vec{v} = \sum_{i=1}^{k} \lambda_i \vec{d_i} \}$$

Par rapport à la définition du cône de dépendance proposée par F. Irigoin [IrTr88b] [3], une contrainte supplémentaire sur λ_i : $\sum_{i=1}^{k} \lambda_i \geq 1$ a été ajoutée. Cette

3. où le cône de dépendance est défini par $\hat{DC} \ = \ \{\vec{v} \mid \exists \vec{d_i} \in D, \exists \lambda_i \geq 0, \lambda_i \vec{d_i}\}$

contrainte garantit que tout ordonnancement linéaire h légal pour D restera légal selon DC. La nécessité de cette contrainte est illustrée sur un exemple 5.1 en chapitre 5.

Une méthode de calcul systématique de DP et DC a été présentée dans le rapport [IrTr87]. Elle consiste à projeter le système de dépendance sur le sous-espace des distances de dépendance, puis à calculer la partie lexico-positive du polyèdre par union des n polyèdres P_i; n est le nombre de boucles imbriquées et P_i est l'intersection du polyèdre initial avec un système de contraintes de lexico-positivité. L'enveloppe convexe des n polyèdres lexico-positifs est égale généralement au **polyèdre de dépendance** DP. Lorsque l'enveloppe convexe de $P_i|_{i=1}^{n}$ est plus grande que l'ensemble des combinaisons convexes de $P_i|_{i=1}^{n}$, DP est composé en m polyèdres lexico-positifs le caractérisant ($m \leq n$). Cet algorithme est détaillé en section 3.3.3.

Exemple 1 :

Calculons maintenant l'abstraction par un polyèdre DP de la dépendance $S1\ \delta^*_{DP}\ S2$ (conflit d'accès aux éléments du tableau T1) dans l'exemple 3.1. En ajoutant l'équation définissant la variable de distance di dans le système de dépendance, on obtient le système de dépendance initial :

$$Sys(i,i',j,di) \ = \ \left\{ \begin{array}{c} i = j \\ 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq i' \leq n \\ di = i' - i \end{array} \right.$$

La projection sur le sous-espace di en utilisant l'algorithme de Fourier-Motzkin donne le système $Sys(di)$.

$$-(n-1) \ \leq \ di \ \leq \ (n-1)$$

L'intersection de ce système avec la condition de lexico-positivité $di \geq 0$ donne :

$$di \ \geq \ 0$$

Ceci définit le polyèdre de dépendance sous la forme d'un système linéaire. Il peut être représenté par le système générateur suivant :

$$\texttt{DP(}dep\texttt{)} = (\ \{0\},\ \{1\},\ \emptyset)$$

Dans cet exemple, `DP(`dep`)` est équivalent à `D(`dep`)` parce que `D(`dep`)` forme naturellement un polyèdre (i.e. toute combinaison convexe des éléments de D est encore une élément de D). Comme, dans cet exemple, la fermeture transitive de DP est égale à DP, DC est défini par :

$$\texttt{DC(}dep\texttt{)} = (\ \{0\},\ \{1\},\ \emptyset)$$

Cependant D est souvent un ensemble discret et n'est donc pas forcément un polyèdre. DP est alors un ensemble plus grand que D. Illustrons ce cas par l'exemple 2.

Exemple 2 :

```
 DO I = 1, n
   DO J = 1, n
S:      T(I+2J) = T(I+4J)
   ENDDO
 ENDDO
```

FIG. 3.2 - *Exemple 3.2*

Calculons le système de dépendance pour les références au tableau `T`: `T(I+4J)` et `T(I+2J)` et calculons les abstractions `D` et `DP`.

Le système de dépendance est :

$$DepSys(i,j,di,dj) \ = \ \begin{cases} i + 4j = i + d_i + 2j + 2d_j \\ 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq i + d_i \leq n \\ 1 \leq j + d_j \leq n \end{cases}$$

Après simplification, on obtient

$$DepSys(i,j,di,dj) \;=\; \left\{ \begin{array}{c} d_i + 2d_j = 2j \\ 1 \leq i \leq n \\ 1 \leq j \leq n \\ 1 \leq i + d_i \leq n \\ 1 \leq j + d_j \leq n \end{array} \right.$$

Ce système étant faisable, il existe bien une dépendance `dep` entre `T(I+4J)` et `T(I+2J)`.

1. Calcul de D :

   En remplacant respectivement `j` dans l'équation par l'ensemble des valeurs que cet indice peut prendre, on obtient `n` équations correspondant aux `n` polyèdres contraignant le vecteur de distance de dépendance (d_i, d_j) :

   $$P_i \;=\; d_i + 2d_j = 2 * i$$

   Puis on fait l'intersection de P_i avec les conditions lexico-positives sur le vecteur de distance de dépendance (d_i, d_j) (définissant la dépendance), qui se traduisent, dans le cas de deux boucles imbriquées, par les deux contraintes suivantes : i) $d_i > 0$, ii) $d_i = 0 \wedge d_j > 0$.

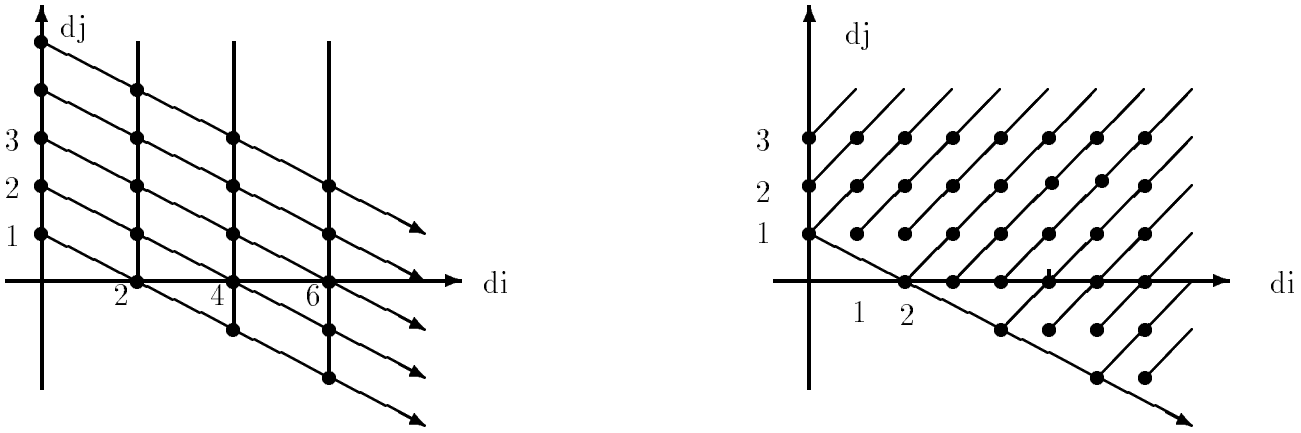
   $$P_i^{posit1} \;=\; \left\{ \begin{array}{c} d_i + 2d_j = 2 * i \\ d_i > 0 \end{array} \right. \quad , \quad P_i^{posit2} \;=\; \left\{ \begin{array}{c} d_i + 2d_j = 2 * i \\ d_i = 0 \\ d_j > 0 \end{array} \right.$$

   La partie lexico-positive de P_i est l'union de P_i^{posit1} et P_i^{posit2}.

   $$P_i^{posit} \;=\; \left\{ \begin{array}{c} d_i + 2d_j = 2 * i \\ d_i \geq 0 \end{array} \right.$$

   Enfin, l'union des $\bigcup_{1 \leq i \leq n} P_i^{posit}$ est effectuée pour obtenir les contraintes définissant D.

   $$\texttt{D(}\textit{dep}\texttt{) = \{ } (d_i, d_j) : \texttt{ (}(d_i \geq 0)\texttt{,} \quad \begin{array}{l} (d_i + 2d_j = 2) \vee \\ (d_i + 2d_j = 4) \vee \\ . \\ . \\ . \\ (d_i + 2d_j = 2n) \end{array} \texttt{ \}}$$



(a) D (dep) (b) DP (dep)

FIG. 3.3 - *D et DP de l'exemple 3.2*

D est illustrée en figure 3.3 à gauche par une série de rayons parallèles. Elle représente un ensemble discret puisqu'ils existent des points entiers (3,0) et (5,0) qui n'appartiennent pas à D.

2. Calcul de DP :
   On projette le système sur le sous-espace (d_i, d_j).

$$DiSys(d_i, d_j) \;=\; \left\{ \; d_i + 2d_j \geq 2 \right.$$

Puis, on fait l'union des intersections avec les conditions lexico-positives. On obtient DP, le polyèdre de dépendance lexico-positif .

$$DiSys^{posit}(d_i, d_j) \;=\; \left\{ \begin{array}{l} d_i + 2d_j \geq 2 \\ d_i \geq 0 \end{array} \right.$$

Sous la forme d'un système générateur,

```
DP(dep) = ( {(0,1)}, {(0,1), (2,-1)} ∅)
```

La partie droite de la figure 3.3 illustre ce polyèdre.

Généralement, DC peut être calculé à partir de DP en ajoutant l'ensemble des sommets de DP à l'ensemble des rayons. L'utilisation de DC par rapport à DP pour **une** dépendance n'a que peu d'intérêt, car DC est moins précis que DP et que son calcul est aussi complexe. Par contre, il permet de représenter toutes les contraintes transitives entre les itérations d'un corps de boucles L :

$$si \quad \vec{i_1} \prec_{d1} \vec{i_2} \quad et \quad \vec{i_2} \prec_{d2} \vec{i_3} \quad alors \quad \vec{i_1} \prec_{d} \vec{i_3} \quad et \quad d = d1 + d2 \in DC(L)$$

Ces contraintes résumées par un seul DC sont nécessaires et suffisantes pour vérifier, par exemple, la légalité du partitionnement de boucles (supernœud [IrTr88a]).

3.3.3 Calcul de DP et DC

Dans cette sous-section, nous détaillons un algorithme qui permet de calculer ces deux abstractions des dépendances.

Algorithme

1. **Calcul du système linéaire de dépendances** $DepSys(\vec{i}, \vec{i'}, \vec{v}, \vec{v'}, \vec{d})$ **:**

$$DepSys(\vec{i}, \vec{i'}, \vec{v}, \vec{v'}, \vec{d}) \;=\; \begin{cases} R_1\ (\vec{i}, \vec{v}) \;=\; R_2\ (\vec{i'}, \vec{v'}) & \textbf{(1)} \\ A_1\ (\vec{i}, \vec{v}) \;\leq\; \vec{0} & \textbf{(2)} \\ A_2\ (\vec{i'}, \vec{v'}) \;\leq\; \vec{0} & \textbf{(3)} \\ \vec{d} \;=\; \vec{i'} - \vec{i} & \textbf{(4)} \\ C_1\ \vec{v} \;=\; b1 & \textbf{(5)} \\ C_2\ \vec{v} \leq b2 & \textbf{(6)} \\ C_1\ \vec{v'} \;=\; b1 & \textbf{(7)} \\ C_2\ \vec{v'} \leq b2 & \textbf{(8)} \\ \vec{v} \;=\; F(i) & \textbf{(9)} \\ \vec{v'} \;=\; F(i') & \textbf{(10)} \end{cases}$$

$\vec{i}$, $\vec{i'}$ sont deux vecteurs d'itérations, $\vec{v}$, $\vec{v'}$ représentent l'ensemble des variables autres que les indices des boucles apparaissant respectivement dans les fonctions d'accès aux éléments du tableau des deux références, $\vec{d}$ est le vecteur de distance de dépendance.

Ce système est composé par :

- un ensemble d'équations caractérisant une dépendance possible ((1)),
- un ensemble d'inéquations définissant les bornes des boucles ((2),(3)),
- un ensemble d'équations définissant le vecteur de distance de dépendance $\vec{d}$ ((4))
- éventuellement, un ensemble de containtes invariantes caractérisant des relations linéaires sur $\vec{v}$ ((5),(6)) et $\vec{v'}$ ((7),(8)), fourni par l'analyse sémantique.
- éventuellement, un ensemble d'équations caractérisant $\vec{v}$ en fonction de $\vec{i}$, fourni par la détection des variables inductives ((9),(10)).

2. si $DepSys(\vec{i}, \vec{i'}, \vec{v}, \vec{v'}, \vec{d})$ est faisable, calcul de la projection[4] $DiSys(\vec{d})$ du système initial sur le sous-espace des distances de dépendance, par l'algorithme de Fourier-Motzkin [Duff74].

$$DiSys(\vec{d}) \;=\; Proj(DepSys,\ \vec{d})$$

3. si le système $DiSys(\vec{d})$ est faisable, calcul de la partie lexico-positive du polyèdre par union des n polyèdres P_i^{posit} où P_i^{posit} est l'intersection de $DiSys(\vec{d})$ avec les conditions de lexico-positivité $PositCond_i$.

$$P^{posit} \;=\; (\cup_{1 \leq i \leq n}(P_i^{posit}))$$

$$P_i^{posit} \;=\; DiSys(\vec{d}) \wedge PositCond_i$$

$$PositCond_i = \begin{cases} d_1 \;=\; 0 \\ d_2 \;=\; 0 \\ \quad \ldots \\ d_i \geq 1 \end{cases}$$

4. en fait, l'enveloppe convexe de la projection

4. **si n=1,** $DP(dep) \;=\; P^{posit}$

5. **si** $n > 1$**, recherche de l'ensemble minimal des polyèdres convexes lexico-positifs (proposition 3.1) caractérisant** P^{posit}

   (a) **rechercher deux polyèdres** P_i^{posit} **et** P_j^{posit}**, dont l'union peut être remplacée par leur enveloppe convexe sans ajouter de points autres que des combinaisons convexes d'eléments de** D**, en utilisant les conditions de la proposition 3.2.**

   - **calculer l'enveloppe convexe des deux polyèdres**

     $$EnvConv(P_i^{posit} \cup P_j^{posit}) \;=\; (\{S_i \cup S_j\}, \{R_i \cup R_j\}, \{D_i \cup D_j\})$$

     où $P_i^{posit} = (\{S_i\},\ \{R_i\},\ \{D_i\})$ **et** $P_j^{posit} = (\{S_j\},\ \{R_j\},\ \{D_j\})$
   - **remplacer** $(P_i^{posit} \cup P_j^{posit})$ **dans l'expression de** P^{posit} **par leur enveloppe convexe.**

   (b) **retourner en 5(a), jusqu'a ce que** P^{posit} **ne puisse plus être simplifié.**

   (c) $DP(dep) \;=\; P^{posit}$

Discussion

1. L'élimination d'une variable d'un système par l'algorithme de Fourier-Motzkin est une méthode de projection rationelle. Le fait que $DiSys(\vec{d})$ ait des solutions entières ne garantit donc pas que $DepSys(\vec{i}, \vec{i'}, \vec{v}, \vec{d})$ ait aussi des solutions entières. Cet algorithme est un test de dépendance en rationnel. Si la projection sur le sous-espace des distances est exacte (i.e. la projection par Fourier-Motzkin est équivalente à la projection entière [Anco91]), DP définit le même ensemble que D . L'exemple 3.1 illustre ce cas.

2. Lorsque D est un ensemble fini, l'enveloppe convexe de D est égale à l'ensemble des combinaisons convexes d'éléments de D. On peut, dans ce cas, utiliser un seul polyèdre (DP) pour caractériser D.

3. DC peut être calculé en ajoutant l'ensemble des sommets de chaque DP_i à l'ensemble des rayons. Si DP est défini par

$$DP(dep) = \{DP_i\} = \{(\{\vec{s_i}\},\ \{\vec{r_j}\},\ \{\vec{d_k}\})\}$$

alors

$$DC(dep) = \{DC_i\} = \{(\{\vec{s_i}\},\ \{\vec{r_j} \cup \vec{s_i}\},\ \{\vec{d_k}\})\}$$

4. L'enveloppe convexe des polyèdres ainsi que DC ne peuvent pas être représentés par une forme unique. Les systèmes résultants peuvent contenir éventuellement des contraintes redondantes. Des algorithmes qui permettent d'éliminer les redondances dans un polyèdre sous la forme de système générateur sont présentés dans [Schr86].

5. Dans l'algorithme présenté, on distingue les cas de dimension 1, des autres cas. Lorsque la dimension est égale à 1, P^{posit} n'est constitué que d'un seul polyèdre lexico-positif, une union serait inutile.

   Dans le cas des dimensions supérieures à 1, on remplace l'union des deux polyèdres lexico-positifs par leur enveloppe convexe si cette enveloppe convexe est égale à l'ensemble des combinaisons convexes des éléments de D. Le maintien de cette condition pour l'abstraction DP est nécessaire pour conserver toutes les informations utiles à l'application des transformations unimodulaires, de partitionnement ou encore au calcul des ordonnancements **multi**-dimensionnels (voir chapitre 4 et 5).

   Par contre, le remplacement de l'union de deux polyèdres lexico-positifs par leur enveloppe convexe, même si cette enveloppe convexe n'est pas lexico-positive, ne fait pas perdre d'ordonnancement **mono**-dimensionnel *valide*. On peut utiliser l'enveloppe convexe des unions de tous les polyèdres lexico-positifs pour calculer un ordonnancement mono-dimensionnel, car l'ensemble des ordonnancements mono-dimensionnels *valides* obtenu à partir de l'enveloppe convexe est identique à l'ensemble des ordonnancements mono-dimensionnels valides calculé à partir de D (voir chapitre 5).

Exemple

Considérons maintenant le programme de l'exemple 3.3. Calculons l'abstraction DP pour la dépendance *dep* d'accès aux éléments du tableau `T`, `T(2*J,M-J)` et `T(2*J,M-J)`.

```
 DO I = 1, n
   DO J = 1, m
     DO K = 1, l
S:       T(2*J,M-J) = X
     ENDDO
   ENDDO
 ENDDO
```

FIG. 3.4 - *Exemple 3.3*

---

1. Calcul du système de dépendances :

$$DepSys(i,j,k,d_i,d_j,d_k) \;=\; \left\{ \begin{array}{c} 2j = 2j + 2dj \\ M - j = M - j - dj \\ 1 \leq i \leq n \\ 1 \leq j \leq m \\ 1 \leq k \leq l \\ 1 \leq i + d_i \leq n \\ 1 \leq j + d_j \leq m \\ 1 \leq k + d_k \leq l \end{array} \right.$$

2. Calcul de la projection sur le sous-espace (d_i, d_j, d_k) :

$$DiSys(d_i,d_j,d_k) \;=\; \left\{ \; d_j = 0 \right.$$

3. Calcul de la partie lexico-positive de $DiSys(d_i,d_j,d_k)$:

$$P_1^{posit} \;=\; \left\{ \begin{array}{l} d_i \geq 1 \\ d_j = 0 \end{array} \right. \quad , \quad P_2^{posit} \;=\; \left\{ \begin{array}{l} d_i = 0 \\ d_j = 0 \\ d_k \geq 1 \end{array} \right.$$

et sous la forme d'un système générateur :

$$P_1^{posit} = (\{(1,0,0)\},\{(1,0,0)\},\{(0,0,1)\}), \quad P_2^{posit} = (\{(0,0,1)\},\{(0,0,1)\},\emptyset)$$

$$P^{posit} = (P_1^{posit} \cup P_2^{posit})$$

4. Recherche de l'ensemble minimal des polyèdres convexes lexico-positifs caractérisant P^{posit}

   (a) – Calcul de l'enveloppe convexe de P^{posit} :

   $$\begin{aligned} EnvConv(P^{posit}) &= (\{(1,0,0),(0,0,1)\},\{(1,0,0),(0,0,1)\},\{(0,0,1)\}) \\ &= (\{(0,0,0)\},\{(1,0,0)\},\{(0,0,1)\}) \end{aligned}$$

   – Comme le niveau de la droite $Level(0,0,1) = 3$ est inférieur au niveau du sommet $Level(0,0,0)$ =4, $EnvConv(P^{posit})$ n'est pas lexico-positive et l'union de P_1^{posit} et P_2^{posit} ne peut être remplacée par son enveloppe convexe.

   (b) P^{posit} n'est peut plus être simplifié. Il est composé des deux polyèdres lexico-positifs $(P_1^{posit} \cup P_2^{posit})$.

   (c)

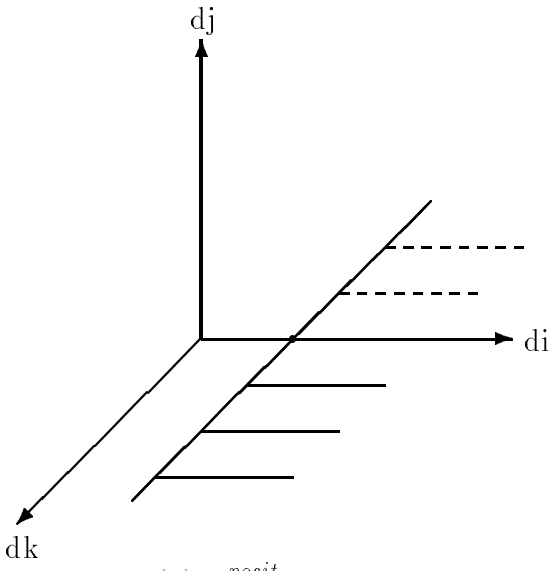
$$DP(dep) = P^{posit} = (DP_1 \cup DP_2)$$

$$DP_1 = P_1^{posit} = (\{(1,0,0)\},\{(1,0,0)\},\{(0,0,1)\})$$

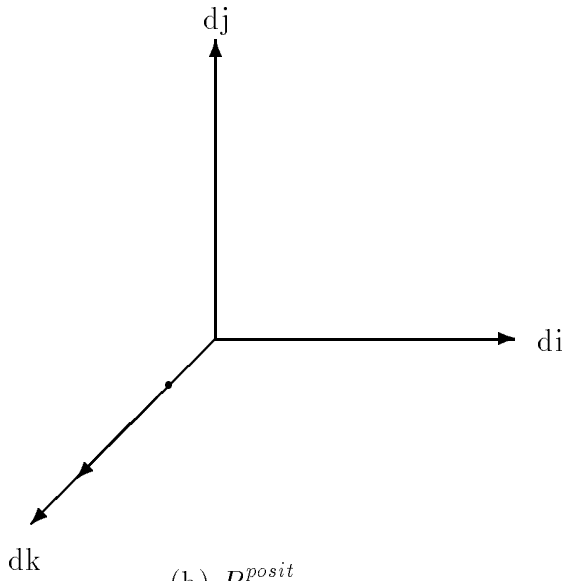
$$DP_2 = P_2^{posit} = (\{(0,0,1)\},\{(0,0,1)\},\emptyset)$$

Conséquence de l'utilisation de l'enveloppe convexe de P^{posit} :
Considérons une transformation unimodulaire définie par la matrice suivante :

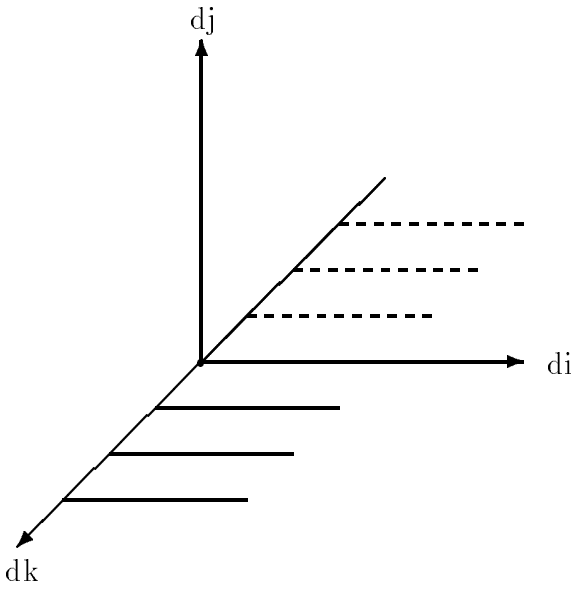
$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



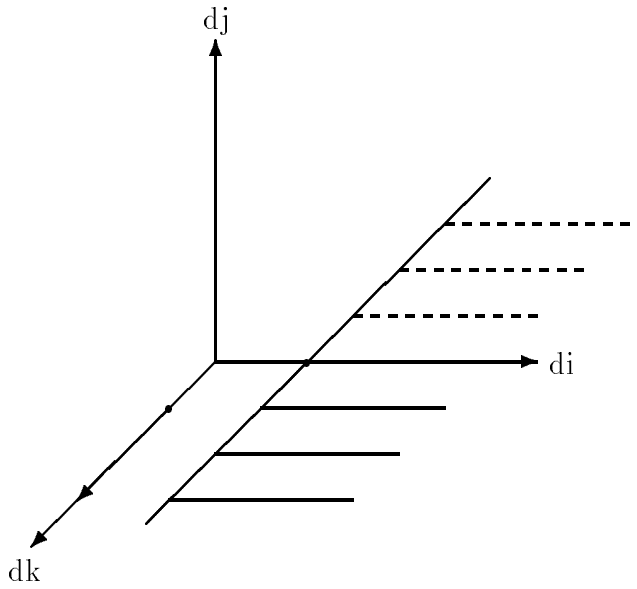
(a) P_1^{posit}



(b) P_2^{posit}



(c) $EnvConv(P^{posit})$



(d) P^{posit}

FIG. 3.5 - *Illustration du calcul de DP pour l'exemple 3.3*

Testons si cette transformation peut être appliquée en utilisant les deux abstractions des dépendances $DP(dep)$ et $EnvConv(DP)$.

- Pour que la transformation définie par M puisse être appliquée selon DP, il faut que la condition $M \times DP(dep) \gg 0$ soit vérifiée. Comme $DP = DP1 \cup DP2$, la condition $(M \times DP_1 \gg 0 \cap M \times DP_2 \gg 0)$ doit être vérifiée.

  Comme $M \times P_1^{posit} = (\{M \times (1,0,0)^t\}, \{M \times (1,0,0)^t\}, \{M \times (0,0,1)^t\})$
  $= (\{(0,1,0)\}, \{(0,1,0)\}, \{(0,0,1)\}) \gg 0$
  et $M \times P_2^{posit} = (\{M \times (0,0,1)^t\}, \{M \times (0,0,1)^t\}, \emptyset) \gg 0$,
  on a bien $M \times DP(dep) \gg 0$.

  M est donc une transformation légale d'après l'abstraction DP.

- Pour que la transformation définie par M puisse être appliquée selon $EnvConv(DP)$, il faut que la condition $M \times EnvConv(DP)(dep) \gg 0$ soit vérifiée. :
  Comme $M \times EnvConv(DP) = (\{M \times (0,0,0)^t\}, \{M \times (1,0,0)^t\}, \{M \times (0,0,1)^t\}) = (\{(0,0,0)\}, \{(0,1,0)\}, \{(0,0,1)\})$
  $\Longrightarrow \neg(M \times EnvConv(DP) \gg 0)$

Si l'on utilise l'abstraction $EnvConv(DP)$, la transformation M ne peut pas être appliquée, alors que légalement elle peut l'être. Cet exemple introduit le problème du choix d'une bonne abstraction des dépendances, contenant suffisamment d'information pour pouvoir appliquer les transformations quand elles peuvent l'être. C'est le sujet que nous étudions dans le chapitre suivant.

3.4 Vecteur de Direction de Dépendance (DDV)

Les abstractions D et DP (ou DC) caratérisent une dépendance par l'ensemble des valeurs que le vecteur de distance $\vec{d}$ prend exactement (D) ou par un sur-ensemble. Ce type d'information basé sur la valeur de $\vec{d}$ est nécessaire pour l'obtention d'un "bon" ordonnancement après une combinaison de transformations modifiant la forme de l'espace d'itérations. Lorsque la forme de l'espace

d'itérations n'est pas modifié par la transformation (e.g. l'échange de boucles, permutation de boucles, l'inversion de boucle), le changement affecte l'ordre ou le signe des éléments de $\vec{d}$. Une autre abstraction conservant un vecteur de signes de $\vec{d}$ appelée *vecteur de la direction de dépendance* (DDV) a donc été proposée. Chaque vecteur de direction peut avoir une valeur parmi $\{+,\ 0,\ -\}$ [Bane88]; pour des raisons historiques, elles sont souvent représentées par $\{<\ ,=,\ >\}$ [Wolf89].

Définition 3.5
$DDV = \{\ d\vec{d}v = (\psi_1, \psi_2, ..., \psi_n) \mid S1(\vec{i})\ \delta^*\ S2(\vec{i'}),\ i_k\ \psi_k\ i'_k\ (1 \leq k \leq n),\ \psi_i \in \{<\ ,=,>\}\}$

D'après la définition 3.5, un DDV est un cône dont les faces sont des hyperplans orthogonaux aux axes des coordonnées. Plusieurs $d\vec{d}v$ élémentaires peuvent se résumer en utilisant les symboles étentus $\{\ \leq, \geq, *, \neq\ \}$ où "*" équivaut aux trois symboles élémentaires $(<, =, >)$, l'absence totale d'information. Nous illustrons par la figure 3.6 toutes les unions possibles dans le cas de deux boucles imbriquées. Le symbole "x" indique que l'union correspondante est soit impossible (i.e. conduit à des ddv's lexico-négatifs); soit imprécise (i.e. impliquerait des ddv's autres que ceux dont on fait l'union).

Une approche hiérarchique du calcul des ddvs traduisant deux dépendances $(S1, R1)\ \delta^*_{ddv_{1\rightarrow 2}}\ (S2, R2)$ et $(S2, R2)\ \delta^*_{ddv_{2\rightarrow 1}}\ (S1, R1)$ aux deux références $R1$, $R2$ des instructions $S1$, $S2$ a été proposée [BuCy86]. Elle est illustrée par la figure 3.7 pour le cas où $S1$ et $S2$ sont imbriquées dans deux boucles .

Dans ce graphe, chaque noeud représente un ddv élémentaire ou un ddv résumé. Au lieu d'appliquer un algorithme de test des dépendances 9 fois, pour tester la faisabilité de l'intersersection du système et d'un ddv afin obtenir tous les ddvs possibles, on calcule les ddvs des sommets vers les feuilles. Lorsque une indépendance a été trouvée en un point, les ddvs en aval ne sont pas calculés. Si Ψ est l'ensemble des ddvs trouvés pour la dépendance de (S1,R1) à (S2,R2). Nous avons :

– les $ddvs \in \Psi$ **lexico-positifs** représentent la dépendance de S1 vers S2;

$\cup$	$(<, <)$	$(<, =)$	$(<, >)$	$(=, <)$	$(=, =)$
$(<, <)$	$(<, <)$	$(<, \leq)$	$(<, \neq)$	$(\leq, <)$	x
$(<, =)$	$(<, \leq)$	$(<, =)$	$(<, \geq)$	x	$(\leq, =)$
$(<, >)$	$(<, \neq)$	$(<, \geq)$	$(<, >)$	x	x
$(=, <)$	$(\leq, <)$	x	x	$(=, <)$	$(=, \leq)$
$(=, =)$	x	$(\leq, =)$	x	$(=, \leq)$	$(=, =)$

FIG. 3.6 - *Unions légales de ddvs élémentaires*

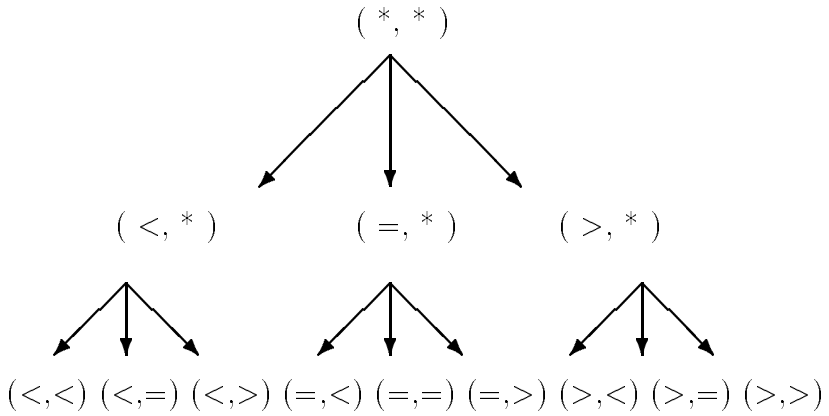


FIG. 3.7 - *Hiérarchie de ddvs*

– les $ddvs \in \Psi$ **lexico-négatifs** représentent la dépendance de S2 vers S1

Les algorithmes de test de dépendance par résolution de système d'inégalités peuvent être utilisés pour calculer cette hiérarchie de ddvs. L'ensemble des ddvs trouvés par un algorithme moins précis sera éventuellement plus grand.

3.5 Profondeur de Dépendance (*Dependance Level* (DL))

La **profondeur de dépendance** a été introduite pour la vectorisation/parallélisation de programmes comportant des boucles imbriquées par Allen & Kennedy [AlKe87]. Elle donne le niveau de la boucle englobante qui doit rester séquentielle pour conserver la lexico-positivité des dépendances à ce niveau et permettre éventuellement la parallélisation de boucles plus internes.

Définition 3.6
$DL = \{k \mid \exists \vec{ddv} = (\psi_1, \psi_2, ..., \psi_n) \in DDV \ t.q. \ \psi_i = 0 \ (1 \leq i \leq k-1) \wedge \psi_k = \text{“}<\text{”}\}$

Une dépendance de profondeur `k` peut être traduite par une série de contraintes sur la variable de distance de dépendance: $d_1 = 0, d_2 = 0, ..., d_{k-1} = 0, d_k > 0$. Ces contraintes peuvent être intégrées au système de dépendance afin de vérifier l'existence possible (système satisfiable) d'une dépendance de profondeur `k`. La profondeur d'une dépendance peut donc être calculée avec les tests acceptant les inéqualités.

3.6 Comparaison des précisions des abstractions

Dans la section précédente, nous avons défini 6 différentes abstractions des dépendances: DI, D, DP, DC, DDV, DL. Nous avons utilisé informellement le concept de “précision” pour comparer une abstraction des dépendances avec les autres. Dans cette section, nous définissons cette précision.

Parmi toutes ces abstractions des dépendances, il n'y a que DI qui énumère précisement toutes les paires d'itérations dépendantes; les autres définissent un

ensemble (exact ou approximatif) des vecteurs de distance de dépendance. Nous définissons un ensemble DI_{apr} approximant DI pour toute abstraction AD autre que DI[5] :

Définition 3.7 *$D_{apr}(AD)$ est égal à l'ensemble des vecteurs de distance de dépendance caractérisant l'abstraction AD.*

Définition 3.8 $DI_{apr}(AD) = \{ (\vec{i}, \vec{i}+\vec{d}) : \vec{i}\ , \vec{i}+\vec{d} \in I^n,\ \vec{d} \in D_{apr}(AD)\}$.

DI ou l'abstraction DI_{apr} définissent explicitement toute les contraintes devant être respectées sur l'ordre d'exécution des instructions. Pour comparer la précison de deux abstractions $AD1$ et $AD2$, on compare les ensembles $DI_{apr}(AD1)$ et $DI_{apr}(AD2)$.

Définition 3.9 *$AD1$ est plus précise que $AD2$ (notée $AD1 \supset AD2$) si $DI_{apr}(AD1) \subseteq DI_{apr}(AD2)$.*

D'après les définitions 3.1, 3.2, DI est inclus dans $DI_{apr}(D)$. La définition 3.9 implique que DI est plus précise que D ($DI \supset D$). En fait, elle est plus précise que toutes les autres abstractions de dépendance par définition. Pour comparer les abstractions AD, autres que DI, on utilisera plutôt les ensembles $D_{apr}(AD)$, plus compacts que $DI_{apr}(AD)$ puisqu'ils sont tous basées sur les vecteurs de distance de dépendance.

Corollaire 3.1 *$AD1$ est plus précise que $AD2$ si $D_{apr}(AD1) \subseteq D_{apr}(AD2)$.*

Preuve :

D'après la définition 3.9, $AD1 \supset AD2$ si $DI_{apr}(AD1) \subseteq DI_{apr}(AD2)$.

D'après la définition 3.8, $DI_{apr}(AD1) \subseteq DI_{apr}(AD2)$ si $D_{apr}(AD1) \subseteq D_{apr}(AD2)$.

Nous en déduisons donc que $AD1 \supset AD2$, si $D_{apr}(AD1) \subseteq D_{apr}(AD2)$

□

5. $DI_{apr}(DI) = DI$

Les relations existant entre D, DP, DC, DDV, DL sont les suivantes : $D_{apr}(D) \subseteq D_{apr}(DP) \subseteq D_{apr}(DC) \subseteq D_{apr}(DDV) \subseteq D_{apr}(DL)$. Nous en déduisons le classement : $D \supset DP \supset DC \supset DDV \supset DL$.

En général, si $AD1 \supset AD2$, $AD2$ peut être calculée à partir de $AD1$. Par contre, l'opération inverse n'est pas possible, car deux dépendances dans $AD1$ peuvent correspondre à une même dépendance dans $AD2$. Une abstraction plus précise permet donc de *mieux* distinguer les dépendances.

Une autre définition de la précision qui découle des précédentes est la suivante :

Définition 3.10 *On dit que AD1 est plus précise que AD2, si les deux conditions suivantes sur les dépendances d_i sont vérifiées :*

$$(1) \quad \exists \ d_1,\ d_2 \quad / \quad AD1(d_1) \mathrel{!}= AD1(d_2) \ \ et \ \ AD2(d_1) = AD2(d_2)$$

$$(2) \quad \forall \ d_3,\ d_4 \ / \ AD1(d_3) = AD1(d_4) \implies AD2(d_3) = AD2(d_4)$$

Elle confirme le classement des differentes abstractions des dépendances : $DI \supset D \supset DP \supset DC \supset DDV \supset DL \supset \perp$ où $\perp$ corrpespond à une information nulle (i.e. $DI_{apr}(\perp) = \{(\vec{i}, \vec{i'}) \mid \vec{i}, \vec{i'} \in I^n\}$).

Exemple :

Considérons maintenant le programme suivant :

```
 DO I = 1, n
   DO J = 1, n
S:        T(I,J) = T(3I,J+1)
   ENDDO
 ENDDO
```

FIG. 3.8 - *Exemple 3.4*

Calculons les abstractions DI, D, DP, DC, DDV et DL pour la dépendance *dep* d'accès aux éléments du tableau `T`: `T(3I,J+1)` et `T(I,J)`

1. calcul de DI :

Le système de dépendance est :

$$DepSys(i,j,i',j') \;=\; \left\{ \begin{array}{c} 3i = i' \\ j+1 = j' \\ 1 \le i \le n \\ 1 \le j \le n \\ 1 \le i' \le n \\ 1 \le j' \le n \end{array} \right.$$

Ce système admet des solutions entières, la dépendance dep existe et est représentée par DI de la façon suivante :

$$DI(dep) = \{((i,j),(3i,j+1)) \mid 1 \le i \le n; 1 \le j \le n\}$$

2. calcul de D :

   Si on introduit les variables de distance di et dj avec : $di = i' - i$; $dj = j' - j$ dans le système initial, on obtient un système équivalent :

$$DepSys(i,j,di,dj) \;=\; \left\{ \begin{array}{c} d_i = 2i \\ d_j = 1 \\ 1 \le i \le n \\ 1 \le j \le n \\ 1 \le i + d_i \le n \\ 1 \le j + d_j \le n \end{array} \right.$$

   En remplacant respectivement `i` dans l'équation par l'ensemble des valeurs que cet indice peut prendre, on obtient

$$D(dep) = \{(2 \times k, 1) \mid 1 \le k \le n\}$$

3. calcul de DP :

   On projette le système sur le sous-espace (d_i, d_j).

$$DiSys(d_i,d_j) \;=\; \left\{ \begin{array}{c} d_i \ge 2 \\ d_j = 1 \end{array} \right.$$

`DP(dep) = ( {(2,1)}, {(1,0)}, ∅)`

4. calcul de DC :

   On calcule DC à partir de DP en ajoutant l'ensemble des sommets de DP à l'ensemble des rayons.

$$\texttt{DC(dep) = (\{(2,1)\}, \{(1,0), (2,1)\}, } \emptyset \texttt{)}$$

5. calcul de DDV :

   Après les intersections de $DiSys(d_i, d_j)$ avec les contraintes traduisant les $ddvs$ possibles, on obtient

$$\texttt{DDV(dep) = \{ (<, <) \}}$$

6. calcul de DL :

   Selon $DDV(dep)$, il est clair que :

$$\texttt{DL(dep) = \{ 1 \}}$$

La figure 3.9 illustre les différentes abstractions des dépendances pour l'exemple 3.4 et met en évidence la précision des différentes abstractions.

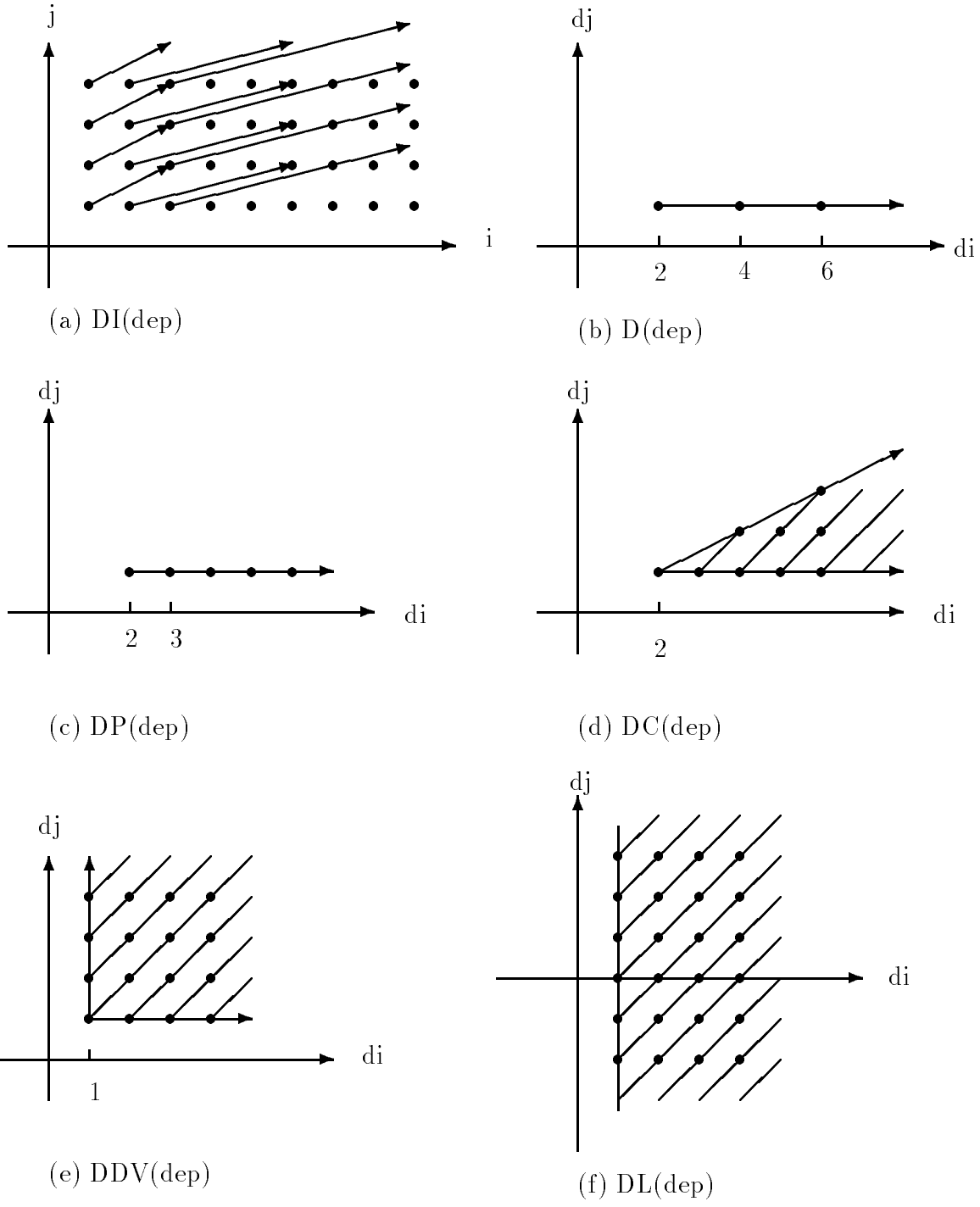


FIG. 3.9 - *Comparaison de précision des abstractions des dépendances*

3.7 Conclusion

Nous avons présenté dans ce chapitre différentes abstractions des dépendances : DI, D, DP, DC, DDV, DL et prouvé que leur précision est décroissante. Nous avons détaillé les algorithmes de calcul du polyèdre de dépendance et du cône de dépendance, qui sont intégrés à l'algorithme du test de dépendance de **PIPS**.

Certains systèmes utilisent des combinaisons de plusieurs abstractions des dépendances. C'est le cas de [WoLa90] et [SaTh92] combinant D et DDV (*dependence vector (DV)*) de la façon suivante : $dv = (d_1, d_2, ..., d_n)$ où $d_i \in Z$ ou $\in (<, >, =, *)$. La dépendance est alors représentée pour chaque dimension par la *distance (D)* si elle est constante, sinon par la *direction (DDV)*. L'abstraction DV est moins précise que la représentation DC pour une dépendance.

Chapitre 4

Abstractions des dépendances et transformations de boucles

Abstractions des dépendances et *transformations de boucles* sont liées . Les dépendances sont utilisées pour vérifier la légalité d'une transformation et une transformation effectuée sur un programme va modifier les relations de dépendance. Puisque les différentes abstractions des dépendances ont des précisions et des coûts de calcul et stockage différents, et parce qu'une transformation pourra être déclarée illégale pour une abstraction des dépendances et pas pour une autre, le choix d'une abstraction des dépendances appropriée à une transformation est important.

Ce chapitre est dédié à l'étude des relations existant entre les différentes abstractions des dépendances et les différentes opérations de transformation de programme. Dans la première section, nous présentons le test de légalité d'une transformation. Les conditions permettant de savoir si une abstraction des dépendances est appropriée à une transformation, c'est à dire contient l'information nécessaire permettant de savoir si la transformation peut être appliquée légalement, sont ensuite décrites. Finalement, nous consacrons la section 3 à l'étude des abstractions des dépendances pour les transformations de réordonnancement, telles que *l'inversion de boucle, la permutation de boucles, les transformations unimodulaires, le partitionnement et la parallélisation.*

Nous introduisons ici quelques notations utilisées au cours du chapitre :

– $TR(L)$: la transformation TR est appliquée à la boucle L;

- TR^{AD} : utilisation de AD comme abstraction de dépendance pour la transformation TR;
- $legal(TR, L) : TR(L)$ est légale;
- $legal(TR, L, AD) : TR^{AD}(L)$ est légale.

4.1 Légalité d'une transformation

Dans le but de mieux exploiter le parallélisme implicite d'un programme, on *transforme* les boucles du programme. Ces transformations réorganisent les instances des instructions des boucles et mettent en évidence le parallélisme implicite. Les dépendances sont utilisées pour tester la légalité de la transformation. Une transformation est **légale** si l'exécution du programme modifié produit le même résultat que l'exécution du programme initial. Autrement dit, le nouvel ordre d'exécution des itérations et des instructions doit conserver toutes les dépendances initiales (c'est à dire préserver la lexico-positivité des dépendances). Nous distinguons, dans la suite de ce chapitre, trois types de transformation de boucles.

Une **transformation de reconstruction** (notée TR_{recons}) de boucles réorganise les instances d'instruction des boucles en conservant toutes les relations de dépendances
(ex. la distribution de boucles). Une **transformation de réordonnancement** (notée TR_{reord}) est une transformation de reconstruction *particulière* qui opère sur les itérations du corps de boucles. C'est une bijection entre deux espaces d'itérations dont les dimensions sont éventuellement différentes (i.e. $\vec{I}^n \Leftrightarrow \vec{J}^m$) (par exemple, le *strip-mining*). Une **transformation unimodulaire** (notée TR_{unimod}) est une transformation de réordonnancement *particulière* qui correspond à une bijection entre deux espaces de **même** dimension. Elle peut être représentée par une matrice (i.e. $\vec{I}^n \Leftrightarrow \vec{J}^n,\ \vec{J}^n = M \times \vec{I}^n$). L'échange de boucles est un exemple de transformation unimodulaire. Les différents types de transformation ont des effets différents sur les instructions et les domaines d'itération des boucles :

- Une transformation de reconstruction transforme l'ensemble des instruc-

tions $S(\vec{i}^n)$ en un nouvel ensemble d'instructions $S'(\vec{j}^m)$ où S' peut être différent de S;

- Une transformation de réordonnancement transforme l'ensemble des instructions $S(\vec{i}^n)$ en un nouvel ensemble d'instructions $S(\vec{j}^m)$ où m peut être différent de n;

- Une transformation unimodulaire transforme l'ensemble des instructions $S(\vec{i}^n)$ en un nouvel ensemble d'instructions $S(\vec{j}^n)$.

La relation d'inclusion existant entre ces différents types de transformation est la suivante : $TR_{recons} \supset TR_{reord} \supset TR_{unimod}$.

Nous présentons dans les trois sections suivantes les tests de légalité pour ces trois types de transformation.

4.1.1 Transformations de reconstruction

La définition 4.1 exprime la condition pour qu'une transformation de reconstruction soit légale. La définition 4.2 décrit ce test de légalité lorsqu'une abstraction des dépendances AD est utilisée.

Définition 4.1 *Une transformation de reconstruction TR(L) est* ***légale*** *et notée* $legal(TR, L)$*, si pour toute dépendance* $S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)$ *existant dans L telle que*

$$TR(S_i(\vec{i}^n)) = S'_i(\vec{j}^m) \quad et \quad TR(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m),$$

alors la condition $S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$ *est vérifiée.*

Cette définition exprime que la lexico-positivité des dépendances doit être préservée. Elle peut être employé si l'abstraction des dépendances exacte DI est utilisée. Pour une autre abstraction AD on utilisera :

Définition 4.2 *Une transformation de reconstruction* $TR^{AD}(L)$[1] *est* ***légale*** *et notée* $legal(TR, L, AD)$ *si pour toute dépendance* $S_i\ \delta^*\ S_j$ *de L telle que*

$$\forall\ (\vec{i}, \vec{i'})\ \in\ DI_{apr}(AD)$$

1. La transformation TR est appliquée au nid de boucles L et l'abstraction des dépendances AD est utilisée pour représenter les dépendances de L

$$et\ \ TR(S_i(\vec{i}^n)) \ = \ S'_i(\vec{j}^m)\ \ et\ \ TR(S_j(\vec{i'}^n)) \ = \ S'_j(\vec{j'}^m),$$

alors la condition $S'_i(\vec{j}^m) \ \prec \ S'_j(\vec{j'}^m)$ *est vérifiée.*

Si une transformation de reconstruction TR est légale d'après la définition 4.2 alors la condition de légalité pour $legal(TR, L)$ de la définition 4.1 est aussi vérifiée; la proposition inverse n'est pas vraie. La condition 4.2 est suffisante mais pas nécessaire pour tester la légalité d'une transformation.

Théorème 4.1 $legal(TR_{recons}, L, AD) \implies legal(TR_{recons}, L)$

Preuve :

On suppose que $TR^{AD}_{recons}(L)$ est légale d'après la définition 4.2, alors

$$\forall\, (\vec{i}, \vec{i'}) \ \in \ DI_{apr}(AD)\ avec\ TR(S_i(\vec{i}^n)) \ = \ S'_i(\vec{j}^m)\ et\ TR(S_j(\vec{i'}^n)) \ = \ S'_j(\vec{j'}^m),$$

la condition $S'_i(\vec{j}^m) \ \prec \ S'_j(\vec{j'}^m)$ est vérifiée.
Comme DI est la représentation des dépendances la plus précise,

$$DI \subseteq DI_{apr}(AD)\ et\ \forall\ (\vec{i}, \vec{i'}) \ \in \ DI \ \ alors \ \ (\vec{i}, \vec{i'}) \in DI_{apr}(AD)$$

Nous déduisons que
$\forall\, (\vec{i}, \vec{i'}) \ \in \ DI\ tel que\ TR(S_i(\vec{i}^n)) \ = \ S'_i(\vec{j}^m)\ et\ TR(S_j(\vec{i'}^n)) \ = \ S'_j(\vec{j'}^m),$
la condition $S'_i(\vec{j}^m) \ \prec \ S'_j(\vec{j'}^m)$ est aussi vérifiée. $\square$

4.1.2 Transformations de réordonnancement

Dans le cas des transformations de réordonnancement, les dépendances entre références $S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)$ correpondent aux dépendances entre itérations $(\vec{i}^n\ \delta^*\ \vec{i'}^n)$. Une dépendance $S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)$ entre deux instructions d'une boucle L sera simplement représentée par une dépendance entre les itérations $\vec{i} \prec_\delta \vec{i'}$. Toutes les dépendances entre les itérations de L peuvent être caractérisées par un ensemble, noté $DI(L)$, contenant toutes les contraintes d'ordonnancement "$\prec_\delta$" entre les itérations (indépendamment des instructions).

$$DI(L) = \{\ (\vec{i_1},\ \vec{i_2})\ : \vec{i_1}\ \prec_\delta\ \vec{i_2} \Leftrightarrow (\exists S_i(\vec{i_1})\ \delta^*\ S_j(\vec{i_2})) \wedge (\vec{i_1} \neq \vec{i_2})\ \}$$

La conservation de toutes les contraintes d'ordonnancement de $DI(L)$ permet de respecter toutes les dépendances entre instructions des boucles.

Les propositions suivantes decrivent les tests de légalité permettant d'appliquer une transformation de réordonnancement à un nid de boucles.

Proposition 4.1 *Une transformation de réordonnancement TR d'un nid de boucles L est* ***légale*** *si et seulement si :*

$\forall\ (\vec{i}^n, \vec{i'}^n)\ \in DI(L)\ \ tel\ que\ \ TR(\vec{i}^n)\ =\ \vec{j}^m\ \ et\ \ TR(\vec{i'}^n)\ =\ \vec{j'}^m$
alors la condition $\vec{j}^m\ \prec\ \vec{j'}^m$ *est vérifiée.*

Preuve :

Une transformation de réordonnancement TR_{reord} étant une transformation de reconstruction particulière, la définition 4.1 s'applique à $TR_{reord} : \forall S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)\ \in\ L\ \ tel\ que$

$$TR_{reord}(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)\ et\ TR_{reord}(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m)\ \text{alors}$$

$$legal(TR_{reord}, L)\ \Longleftrightarrow\ \forall S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n) \in L\ , S'_i(\vec{j}^m)\ \prec\ S'_j(\vec{j'}^m)$$

Comme il s'agit d'une transformation de réordonnancement, $S_i\ et\ S_j$ appartiennent au même nid de boucles imbriquées et $S'_j = S_j$, $S'_i = S_i$.
Nous savons que

$$S_i(\vec{i}^m)\ \prec\ \ S_j(\vec{i'}^m)\ \Longleftrightarrow\ (\vec{i}^m\ \prec\ \vec{i'}^m) \vee ((\vec{i}^m = \vec{i'}^m) \wedge (S_i\ avant\ S_j)).$$

$$et\ S_i(\vec{j}^m)\ \prec\ S_j(\vec{j'}^m)\ \Longleftrightarrow\ (\vec{j}^m\ \prec\ \vec{j'}^m) \vee ((\vec{j}^m = \vec{j'}^m) \wedge (S_i\ avant\ S_j)).$$

Lorsque $\vec{i}^n = \vec{i'}^n$, après réordonancement,
la contrainte $(\vec{j}^m = \vec{j'}^m) \wedge (S_i\ avant\ S_j)$ est toujours vérifiée. On a donc :

$$legal(TR_{reord}, L)\ \Longleftrightarrow\ \forall\ S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n),\ \ \vec{i}^n \neq \vec{i'}^n\ alors\ \vec{j}^m\ \prec\ \vec{j'}^m$$

Ceci est équivalent à

$$legal(TR_{reord}, L)\ \Longleftrightarrow\ \forall\ \vec{i}^n\ \prec_\delta \vec{i'}^n\ alors\ \vec{j}^m\ \prec\ \vec{j'}^m$$

$$legal(TR_{reord}, L)\ \Longleftrightarrow\ \forall\ (\vec{i}^n, \vec{i'}^n) \in DI(L)\ alors\ \vec{j}^m\ \prec\ \vec{j'}^m \qquad \square$$

Cette proposition peut être employée si l'abstraction des dépendances $DI(L)$ est utilisée. Pour une autre abstraction AD moins précise que $DI(L)$ on utilisera :

Proposition 4.2 *Une transformation de réordonnancement $TR^{AD}(L)$ est* ***légale*** *si et seulement si:*

$$\forall\ \vec{i}^n \prec_\delta \vec{i'}^n \in DI_{apr}(AD)(L)\ tel\ que\ TR(\vec{i}^n) = \vec{j}^m\ et\ TR(\vec{i'}^n) = \vec{j'}^m,$$

alors la condition $\vec{j}^m \prec \vec{j'}^m$ est vérifiée.

Preuve :

Une transformation de réordonnancement TR_{reord} étant une transformation de reconstruction particulière, la définition 4.2 s'applique à TR_{reord} :

$legal(TR_{reord}, L, AD) \iff$
$\forall\ (dep = S_i\ \delta^*\ S_j),\ \forall(\vec{i}^n, \vec{i'}^n) \in DI_{apr}(AD)(dep)\ alors\ S_i(\vec{j}^m) \prec S_j(\vec{j'}^m)$

Puisque toutes les dépendances intra-itération $\vec{i}^n = \vec{i'}^n$ sont naturellement respectées par une transformation de réordonnancement, on peut les ignorer. On obtient donc pour $\vec{i}^n \neq \vec{i'}^n$:

$$S_i(\vec{j}^m) \prec S_j(\vec{j'}^m) \iff \vec{j}^m \prec \vec{j'}^m$$

D'où $legal(TR_{reord}, L, AD) \iff$
$\forall\ (dep = S_i\ \delta^*\ S_j),\ \forall(\vec{i}^n, \vec{i'}^n) \in DI_{apr}(AD)(dep),\ \vec{i}^n \neq \vec{i'}^n\ alors\ \vec{j}^m \prec \vec{j'}^m$

Ceci est équivalent à

$legal(TR_{reord}, L, AD) \iff \forall\ \vec{i}^n \prec_\delta \vec{i'}^n \in DI_{apr}(AD)(L)\ alors\ \vec{j}^m \prec \vec{j'}^m$ □

Théorème 4.2

$$legal(TR_{reord}, L, AD) \implies legal(TR_{reord}, L)$$

Preuve :

Sachant que
$legal(TR_{reord}, L, AD) \iff \forall\ \vec{i}^n \prec_\delta \vec{i'}^n \in DI_{apr}(AD)(L)\ alors\ \vec{j}^m \prec \vec{j'}^m$

et que $DI(L) \subseteq DI_{apr}(AD)(L)$

On a

$\forall \vec{i}^{\,n} \prec_\delta \vec{i}'^{\,n} \in DI_{apr}(AD)(L)\, alors\, \vec{j}^{\,m} \prec \vec{j}'^{\,m} \Longrightarrow \forall \vec{i}^{\,n} \prec_\delta \vec{i}'^{\,n} \in DI(L)\, alors\, \vec{j}^{\,m} \prec \vec{j}'^{\,m}$

D'où :

$$legal(TR_{reord}, L, AD) \implies legal(TR_{reord}, L) \qquad \square$$

4.1.3 Transformations unimodulaires

Une transformation unimodulaire est une bijection entre deux espaces d'itération de **même** dimension I^n et I'^n. Elle peut être caractérisée par une matrice unimodulaire M où $\vec{i'} = M \times \vec{i}$. La condition pour qu'une dépendance entre deux itérations (i_1, i_2) soit préservée après une transformation unimodulaire où $i'_1 = M \times i_1,\ i'_2 = M \times i'_2$ est que $i'_2 \gg i'_1$. La même relation de bijection existe entre les deux espaces de distances de dépendance D^n et D'^n correpondant. Si on définit $\vec{d} = i_2 - i_1$ et $\vec{d'} = i'_2 - i'_1$, alors $\vec{d'} = M \times \vec{d}$ et

$$i'_2 \gg i'_1 \iff \vec{d'} \gg 0$$

Pour une transformation unimodulaire, la préservation de la lexico-positivité des dépendances selon D traduit donc aussi la conservation de la lexico-positivité des dépendances entre les itérations de DI.

On peut utiliser un seul ensemble, noté $D(L)$, pour caractériser tous les vecteurs de distance des dépendances entre les itérations du nid de boucles (indépendamment des instructions).

$$D(L) = \{\ \vec{d}\ : \vec{d} = \vec{i_2} - \vec{i_1} \Leftrightarrow (\vec{i_1}\ \delta^*\ \vec{i_2}\ ) \wedge (\vec{i_1} \neq \vec{i_2})\}$$

Les propositions suivantes decrivent les tests de légalité permettant d'appliquer une transformation unimodulaire à un programme.

Proposition 4.3 *Une transformation unimodulaire TR d'une boucle L est* ***légale*** *si et seulement si :*

$$\forall\ \vec{d^n} \in D(L)\ \ /\ \ \ TR_{unimod}(\vec{d^n})\ =\ \vec{d'}^{\,n}$$

alors la condition $\vec{d'}^{\,n} \gg 0$ est vérifiée.

Preuve :

Une transformation unimodulaire TR_{unimod} étant une transformation de réordonnancement particulière, la proposition 4.2 s'applique à TR_{unimod} :
$\forall\ (\vec{i}^{\,n}, \vec{i}^{\,n}) \in DI(L)\ \ tel\ que\quad TR(\vec{i}^{\,n}) = \vec{j}^{\,m}\ \ et\ \ TR(\vec{i}^{\,n}) = \vec{j}^{\,\prime m}$
alors

$$legal(TR_{unimod}, L) \iff \forall(\vec{i}^{\,n}, \vec{i}^{\,n}) \in DI(L)\ , alors\ \vec{j}^{\,m} \prec \vec{j}^{\,\prime m}$$

Comme il s'agit d'une transformation unimodulaire, la dimension de l'espace d'itérations ne change pas après transformation et le changement de base peut être caractèrisé par une matrice unimodulaire M telle que :

$$m = n \quad et \quad \vec{j}^{\,m} = M \times \vec{i}^{\,n}\ et\ \vec{j}^{\,\prime m} = M \times \vec{i}^{\,\prime n}$$

Nous savons que
$\vec{d}^{\,n} \in D(L) \iff \exists\ (\vec{i}^{\,n}, \vec{i}^{\,n}) \in DI(L)\ \ avec\ \ \vec{d}^{\,n} = \vec{i}^{\,n} - \vec{i}^{\,n}$
et $TR_{unimod}(\vec{d}^{\,n}) = \vec{d}^{\,\prime n} = M \cdot \vec{d}^{\,n} = \vec{j}^{\,\prime m} - \vec{j}^{\,m}$. D'où $\vec{j}^{\,m} \prec \vec{j}^{\,\prime m} \iff \vec{d}^{\,\prime n} \gg 0$

On a l'équivalence :
$\forall(\vec{i}^{\,n}, \vec{i}^{\,n}) \in DI(L)\, alors\, \vec{j}^{\,m} \prec \vec{j}^{\,\prime m} \iff \forall\, \vec{d}^{\,n} \in D(L)\ /\ TR_{unimod}(\vec{d}^{\,n}) = \vec{d}^{\,\prime n}$
alors $\vec{d}^{\,\prime n} \gg 0$
Ce qui équivaut à
$legal(TR_{unimod}, L) \iff \forall\ \vec{d}^{\,n} \in D(L)\ \ tel\ que\ \ TR_{unimod}(\vec{d}^{\,n}) = \vec{d}^{\,\prime n}$
$alors\ \vec{d}^{\,\prime n} \gg 0$ $\square$

Proposition 4.4 *Une transformation unimodulaire $TR^{AD}(L)$ est **légale** si et seulement si :*

$$\forall\ \vec{d}^{\,n} \in D_{apr}(AD)\ \ /\ \ TR_{unimod}(\vec{d}^{\,n}) = \vec{d}^{\,\prime n}$$

alors la condition $\vec{d}^{\,\prime n} \gg 0$ est vérifiée.

Preuve :

Une transformation unimodulaire TR_{unimod} étant une transformation de réordonnancement particulière, la proposition 4.2 s'applique à TR_{unimod} :
$\forall\ (\vec{i}^{\,n}, \vec{i}'^{\,n})\ \in DI_{apr}(L)\ \ tel\ que\ \ \ TR(\vec{i}^{\,n})\ =\ \vec{j}^{\,m}\ \ et\ \ TR(\vec{i}'^{\,n})\ =\ \vec{j}'^{\,m}$
alors

$$legal(TR_{unimod}, L, AD)\ \Longleftrightarrow\ \forall(\vec{i}^{\,n}, \vec{i}'^{\,n})\ \in DI_{apr}(L)\ alors\ \vec{j}^{\,m}\ \prec\ \vec{j}'^{\,m}$$

Comme il s'agit d'une transformation unimodulaire, la dimension de l'espace d'itérations ne change pas après transformation et le changement de base peut être caractèrisé par une matrice unimodulaire M :

$$m = n\quad et\quad \vec{j}^{\,m} = M \times \vec{i}^{\,n}\ et\ \vec{j}'^{\,m} = M \times \vec{i}'^{\,n}$$

Comme $\vec{d}^{\,n} \in D_{apr}(L)\ \Longleftrightarrow\ \exists\ (\vec{i}^{\,n}, \vec{i}'^{\,n})\ \in DI_{apr}(L)\ \ avec\ \ \vec{d}^{\,n} = \vec{i}'^{\,n} - \vec{i}^{\,n}$
et $TR_{unimod}(\vec{d}^{\,n})\ =\ \vec{d}'^{\,n}\ = M \cdot \vec{d}^{\,n}\ =\ \vec{j}'^{\,m} - \vec{j}^{\,m}$.
Alors, $\vec{j}^{\,m}\ \prec\ \vec{j}'^{\,m}\ \Longleftrightarrow\ \vec{d}'^{\,n} \gg 0$

On a :
$\forall(\vec{i}^{\,n}, \vec{i}'^{\,n})\ \in DI_{apr}(L)\,, alors\ \vec{j}^{\,m}\ \prec\ \vec{j}'^{\,m}\ \Longleftrightarrow\ \forall\ \vec{d}^{\,n} \in D_{apr}(L)\ /\ TR_{unimod}(\vec{d}^{\,n})\ =\ \vec{d}'^{\,n}$
alors $\vec{d}'^{\,n} \gg 0$
Ceci est équivalent à :
$legal(TR_{unimod}, L, AD)\ \Longleftrightarrow\ \forall\ \vec{d}^{\,n} \in D_{apr}(L)\ \ tel\ que\ \ TR_{unimod}(\vec{d}^{\,n})\ =\ \vec{d}'^{\,n}$
$alors\ \vec{d}'^{\,n} \gg 0$ $\quad\square$

Théorème 4.3

$$legal(TR_{unimod}, L, AD)\ \Longrightarrow\ legal(TR_{unimod}, L)$$

Preuve :

Sachant que :

$legal(TR_{unimod}, L, AD) \iff \forall \vec{d}^n \in D_{apr}(AD) \ / \ TR_{unimod}(\vec{d}^n) = \vec{d'}^n \ alors \ \vec{d'}^n \gg 0$
et que $D(L) \subseteq D_{apr}(AD)(L)$.

On a $\forall \vec{d}^n \in D_{apr}(AD) \ / \ TR_{unimod}(\vec{d}^n) = \vec{d'}^n \ alors \ \vec{d'}^n \gg 0$
$\Longrightarrow \quad \forall \ \vec{d}^n \in D(L) \ / \ TR_{unimod}(\vec{d}^n) = \vec{d'}^n \ alors \ \vec{d'}^n \gg 0$

D'où : $legal(TR_{unimod}, L, AD) \implies legal(TR_{unimod}, L)$ $\square$

4.2 Abstraction appropriée à une transformation

Rappellons l'ordre de précisions des 6 différentes abstractions des dépendances :
$DI \supset D \supset DP \supset DC \supset DDV \supset DL$ (voir la section 3.6). Dans les propositions précédentes, nous avons proposé l'utilisation des répresentations DI ou D ($DI_{apr}(AD)$ ou $D_{apr}(AD)$ pour une abstraction AD) pour tester la légalité des transformations. Cependant l'information contenue dans chacune de ces représentations de dépendances est parfois trop "riche" pour tester la légalité d'une transformation. Elles contiennent plus d'information que nécessaire au test de légalité, et une représentation moins précise, plus facile à calculer, peut être utilisée.

Nous introduisons maintenant les définitions et théorèmes qui permettent de définir les abstractions *admissibles* et *minimales* permettant de tester la légalité d'une transformation TR.

Définition 4.3 *Soit AD, une abstraction moins précise que DI, et TR une transformation de reconstruction. Si pour tout nid de boucles L,*

$$legal(TR, L, DI) \implies legal(TR, L, AD)$$

(ou $\neg legal(TR, L, AD) \implies \neg legal(TR, L, DI)$*)*

alors l'abstraction AD est admissible pour tester la légalité de la transformation TR.

Afin de prouver qu'une abstraction des dépendances AD est admissible pour tester la légalité d'une transformation de reconstruction, on compare les résultats des tests de légalité obtenus selon DI et AD. Pour une transformation unimodulaire, on compare les résultats des tests de légalité obtenus selon D et AD.

Théorème 4.4 *Soit AD, une abstraction moins précise que D, et TR une transformation unimodulaire. Si pour tout nid de boucles L,*

$$legal(TR, L, D) \implies legal(TR, L, AD)$$

(ou $\neg legal(TR, L, AD) \implies \neg legal(TR, L, D)$*)*

alors l'abstraction AD est admissible pour tester la légalité de la transformation TR.

Preuve :

D'après la définition 4.3 et la proposition 4.3,

$$legal(TR_{unimod}, L, DI) \iff legal(TR_{unimod}, L, D)$$

D est donc une abstraction admissible pour les transformations unimodulaires.

Soit $D \supset AD$ tel que

$$legal(TR_{unimod}, L, D) \implies legal(TR_{unimod}, L, AD)$$

Nous obtenons que :

$$legal(TR_{unimod}, L, DI) \implies legal(TR_{unimod}, L, AD)$$

D'après la définition 4.3, l'abstraction AD est donc admissible pour tester la légalité de la transformation TR_{unimod}. $\square$

Les abstractions admissibles pour chacune des transformations sont parfois multiples. En fait, toute abstraction plus précise qu'une abstraction admissible est aussi admissible pour cette transformation.

Théorème 4.5 *Soit AD une abstraction des dépendances admissible pour tester la légalité d'une transformation TR, alors toute abstraction AD_i / $AD_i \supset AD$, est aussi admissible pour tester la légalité de la transformation TR.*

Preuve :

Sachant que AD est admissible pour $TR(L)$ et que $AD_i \supset AD$.
D'après la définition 4.3,

$$legal(TR, L, DI) \implies legal(TR, L, AD)$$

D'après la définition 4.2,

$legal(TR, L, AD) \iff \forall\, (\vec{i}, \vec{i'}) \in DI_{apr}(AD)\ avec\ TR(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)\ et\ TR(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m),\ alors S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$.

Comme $AD_i \supset AD$,

$$\forall\, (\vec{i}, \vec{i'}) \in DI_{apr}(AD_i)\ \ alors\ \ (\vec{i}, \vec{i'}) \in DI_{apr}(AD)$$

Nous déduisons que
$\forall\, (\vec{i}, \vec{i'}) \in DI_{apr}(AD_i)\ avec\ TR(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)\ et\ TR(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m),$
$alors\ \ S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$

Ceci est équivalent à $legal(TR, L, AD) \implies legal(TR, L, AD_i)$.
D'où $legal(TR, L, DI) \implies legal(TR, L, AD_i)$.
D'après la définition 4.3, AD_i est donc admissible pour tester la légalité de la transformation TR. $\square$

Puisqu'il existe éventuellement plusieurs abstractions admissibles pour une transformation, il est intéressant de trouver celle d'entre elles qui a la précision *minimale*, c'est à dire la précision nécessaire minimale pour pouvoir tester si la transformation peut être appliquée légalement. Nous définissons ci-dessous une *abstraction minimale*.

Définition 4.4 *Soit $AD1$ une abstraction admissible pour la transformation TR et $AD2$ telle que ($AD1 \supset AD2$ et $\neg\exists AD3 \ / \ AD1 \supset AD3 \supset AD2$).*
Si $\exists L$ telle que $legal(TR, L, AD1) \wedge \neg(legal(TR, L, AD2))$ alors l'abstraction $AD1$ est minimale pour tester la légalité de la transformation TR.

Nous étudions, dans la section suivante, les abstractions des dépendances appropriées pour les transformations de réordonnancement telles que: l'inversion de boucle, la permutation de boucles, les transformations unimodulaires, le partitionnement et la parallélisation.

4.3 Abstractions appropppriées aux transformations de réordonnancement

Pour réordonnancer les itérations d'une boucle, on doit conserver toutes les dépendances entre itérations $\vec{i_1} \prec \vec{i_2}$., l'itération $\vec{i_1}$ doit être ordonnancée avant l'itération $\vec{i_2}$. Les transformations de réordonnancement ne prennent en compte que les dépendances sur les itérations, nous nous intéressons donc, dans cette section, uniquement aux contraintes caractérisant les itérations dépendantes, indépendamment des instructions. Cette particularité des transformations de réordonnancement nous permet de représenter toutes les dépendances des itérations par un seul ensemble de dépendances $AD(L)$ qui est égal à l'union de toutes les dépendances entre instructions représentées par AD .

Nous présentons maintenant les opérations permettant de faire l'union des dépendances existant au sein du nid de boucles pour chaque abstraction des dépendances : $DI(L)$, $D(L)$, $DP(L)$, $DC(L)$, $DDV(L)$, et $DL(L)$. On suppose, dans la suite, qu'un nid de boucles L porte `k` dépendances dep_i, $AD(dep_i)$ correspond à l'abstraction par AD de la i-ème dépendance.

(1) $DI(L) = \{\cup_{1 \leq i \leq k} DI(dep_i)\}$

(2) $D(L) = \{\cup_{1 \leq i \leq k} D(dep_i)\}$

(3) $DDV(L) = \{\cup_{1 \leq i \leq k} DDV(dep_i)\}$

(4) $DL(L) = \{\cup_{1 \leq i \leq k} DL(dep_i)\}$

(5) $DP(L) = (\cup_j DP_j)_{1 \leq j \leq n}$ / $n \leq k$, et $\cup_j DP_j \subseteq Conv(\cup_{1 \leq i \leq k} DP(dep_i))$

(6.1) $DC(L) = (\cup_j DC_j)_{1 \leq j \leq n}$ / $n \leq k$, et $\cup_j DC_j \subseteq Conv(\cup_{1 \leq i \leq k} DC(dep_i))$

(6.2) $DC(L) = (\cup_j DC_j)$ si $DP(L) = (\cup_j DP_j)$ avec

$$DC_j = (\{S_j^{dpj}\},\ \{\{R_j^{dpj} \cup \{S_j^{dpj}\}\},\ \{D_j^{dpj}\}) \text{ et}$$

$$DP_j = (\{S_j^{dpj}\},\ \{R_j^{dpj}\},\ \{D_j^{dpj}\})$$

L'union des dépendances dep_i représentées par DI, D ou DL est égale à l'union des éléments apartenant à chacun des ensembles $DI(dep_i)$, $D(dep_i)$ et $DL(dep_i)$.

Les unions légales de dépendances représentées par des DDV sont illustrées par la figure 3.6 dans la section 3.4.

L'union des dépendances dep_i représentées par deux DPs ou deux DCs peut être remplacée par leur enveloppe convexe sous condition que l'enveloppe convexe des dépendances soit égale à l'ensemble des combinaisons convexes de ses éléments. Lorsque l'enveloppe convexe de l'union des polyèdres contient des éléments qui ne sont pas des combinaisons convexes, $DP(L)$ et $DC(L)$ sont composés de plusieurs petits polyèdres lexico-positifs.

$DC(L)$ peut se calculer de deux manières différentes, par l'union de $DC(dep_i)(1 \leq i \leq k)$ et par $DP(L)$.

Les polyèdres résultant du calcul de $DP(L)$ et $DC(L)$ possèdent des éléments redondants. Des algorithmes qui permettent d'éliminer les redondances dans un polyèdre sous la forme de système générateur sont présentés dans [Schr86].

Nous avons les mêmes relations entre les éléments représentés par $D(L)$,

$DP(L)$ et $DC(L)$ que celles qui existent entre $D(dep)$, $DP(dep)$ et $DC(dep)$. Plus précisément, les éléments de $DP(L)$ sont des combinaisons linéaires convexes de $D(L)$ et ceux de $DC(L)$ des combinaisons positives linéaires des éléments de $D(L)$.

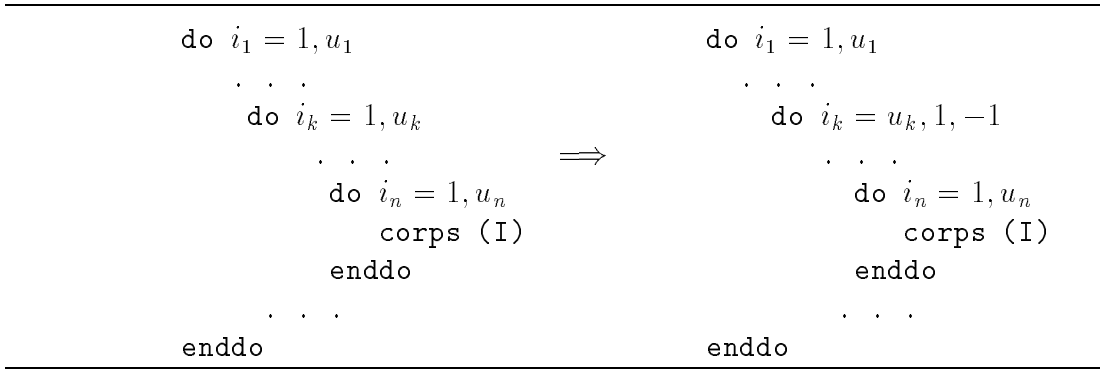
Nous présentrons maintenant les quatre transformations de réordonnancement les plus classiques : *l'inversion de boucle, la permutation de boucles, la transformation unimudulaire, le partitionnement et la parallélisation.* Pour chacune d'elles nous présenterons : sa déscription, son effet sur le vecteur de distance $\vec{d}$, l'abstraction des dépendances la plus appropriée et le test de légalité associé.

4.3.1 Inversion de boucle (*loop reversal*)

L'inversion de boucle inverse simplement l'ordre d'exécution des itérations d'une boucle. C'est une transformation unimodulaire élémentaire.

Spécification :

$Invers_K\ (l_1, l_2, ..., l_n)$ inverse la boucle l_k.



Effet de la transformation sur $\vec{d}$:

$$si\ \vec{d} = (d_1, ..., d_k, ..., d_n),\ Invers_K\ (\vec{d}) = (d_1, ..., -d_k, ..., d_n)$$

Test de légalité :

$$legal(Invers_K,\ L) \iff \forall \vec{d} \in D(L), (d_1, ..., -d_k, ..., d_n) \gg 0$$

L'abstraction admissible et minimale pour pouvoir effectuer légalement cette transformation est : DL

Preuve :

1. DL admissible :

   Supposons qu'il existe une boucle L telle que $Invers^{DL}{}_{K}\ (L)$ soit illégale. Alors :

   $\forall k \in DL(L)\ \ ,\ \ \exists \vec{d} \in D(L)\ \ /\ \ \vec{d} = (0,0,...,0,d_k,*,...,*)\ (d_k > 0)$

   et $\quad \vec{d'} = Invers_{\ K}\ (\vec{d}) = (0,0,...,0,-d_k,*,...,*) \ll 0$

   $\vec{d'} \ll 0 \implies Invers^{D}{}_{K}\ (L)$ est illégale.

   Comme $D \supset DL$, DL est admissible pour l'*inversion de boucle*.

2. DL minimale :

   Sachant que DL est admissible pour l'*inversion de boucle*. Nous montrons ici que l'abstraction DL est aussi minimale pour tester la légalité de l'inversion de boucle, en prenant un exemple pour lequel la légalité est donnée par DL et indéterminée pour l'abstraction moins précise $\perp$ moins précise que DL [2].

   Soit L une boucle telle que $DL(L) = \{\ p\ \}$, $p \neq k$

   (a) $DL(L) = \{p\} \implies \forall \vec{d} \in D(L), \vec{d} = (0,0,...,0,d_p,*,...,*)\ (d_p > 0)$
   $p \neq k \implies Invers_{\ K}\ (\vec{d}) = (0,0,...,0,d_p,*,...,*) \gg 0$
   $\implies Invers\ {}^{DL}_{K}\ (L)$ est bien légale

   (b) Pour $\perp$, nous n'avons aucune information sur $\vec{d}$, la k-ième composante de $\vec{d}$ pourrait être positive. Il pourrait exister un vecteur $\vec{d} \in D(L)$,
   $\vec{d} = (0,0,...,0,d_k,*,...,*)\ (d_k > 0)$ et $Invers_{\ K}\ (\vec{d}) \ll 0$.
   $\implies \quad Invers^{\perp}{}_{K}\ (L)$ est illégale.

   Puisque $DL \supset \perp$, DL est minimale pour l'*inversion de boucle*.

Test de légalité associé à l'abstraction minimale DL :

$$legal(Invers_{\ K},\ L) \iff k \notin DL(L)$$

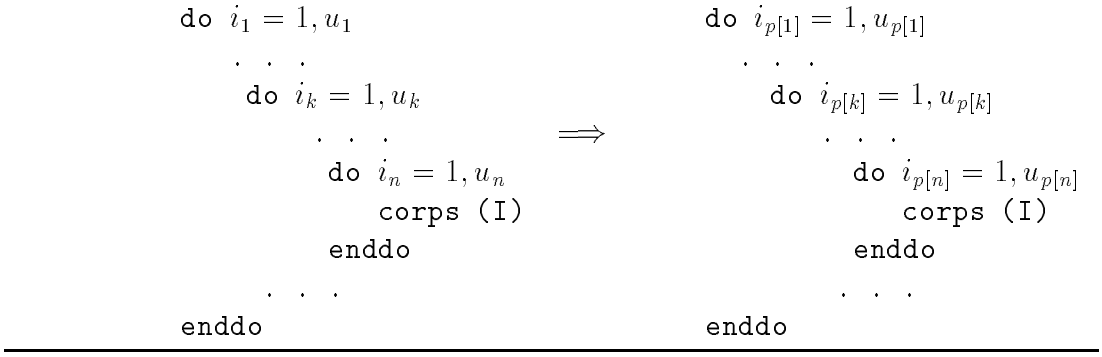
2. $\perp$ signifie que l'on ne dispose d'aucune information

4.3.2 Permutation de boucles

La permutation de boucles échange l'ordre d'exécution des boucles. C'est aussi une transformation unimodulaire élémentaire. L'échange de boucles est une permutation particulière qui n'echange que deux boucles.

Spécification :

$Perm\ _P\ (L)$ effectue une permutation P des boucles de L, $L' = (l_{p[1]}, l_{p[2]}, ..., l_{p[n]})$



Effet de la transformation sur $\vec{d}$:

$$si\ \vec{d} = (d_1, ..., d_k, ..., d_n),\ \vec{d'} = Perm\ _P\ (\vec{d}) = (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]})$$

Test de légalité :

$$legal(Perm\ _P,\ L) \iff \forall \vec{d} \in D(L),\ (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]}) \gg 0$$

L'abstraction admissible et minimale : pour pouvoir effectuer légalement cette transformation est : DDV

Preuve :

1. DDV admissible :
   Supposons qu'il existe une permutation P telle que $Perm_P^{DDV}(L)$ soit illégale.
   Alors $\exists d\vec{d}v \in DDV(L)\ \ /\ \ Perm\ _P\ (d\vec{d}v) \ll 0$.

$\Longrightarrow \exists \vec{d} \in D(L) \ / \ \psi(\vec{d}) = d\vec{d}v \ et \ Perm_P(\psi(\vec{d})) = Perm_P(d\vec{d}v) \ll 0$

où

$$\psi(x) = \begin{cases} < & si\ x > 0 \\ > & si\ x < 0 \\ = & si\ x = 0 \end{cases}$$

$Comme \ \ Perm_P(\psi(\vec{d})) = \psi(Perm_P(\vec{d})) \Longrightarrow Perm_P(\vec{d}) \ll 0$

$\Longrightarrow Perm_P^D(L)$ est illégale

Comme $D \supset DL$, DDV est donc admissible pour la *permutation de boucles*.

2. DDV minimale :

   Sachant que DDV est admissible pour toute *permutation de boucles* et que $DDV \supset DL$, nous montrons ici que l'abstraction DDV est aussi minimale pour tester la légalité de la permutation de boucles, en prenant un exemple pour lequel la légalité est donnée par DDV et indéterminée pour une abstraction moins précise DL.

   Soit $L = (l_1, l_2, l_3)$ un nid de boucles tel que :

   i) $DDV(L) = \{(<,=,>)\ \}$

   ii) $DL(L) = \{\ 1\ \}$

   Soit P la matrice de permutation, P=(2,1,3).

   (a) $DDV(L) = \{(<,=,>)\} \Longrightarrow$
   $\forall \vec{d} = (d_1, d_2, d_3) \in D(L) : d_1 > 0, d_2 = 0, d_3 < 0.$
   $Perm(\vec{d}) = (d_2, d_1, d_3) \gg 0 \Longrightarrow Perm_P^{DDV}(L)$ est légale

   (b) $DL(L) = \{\ 1\ \} \Longrightarrow \exists \vec{d} \in D(L) \ / \ \vec{d} = (d_1, *, *) \ \ et \ \ d_1 > 0.$
   $Perm(\vec{d}) = (*, d_1, *)$
   N'ayant aucun renseignement sur la première composante de $Perm(\vec{d})$, qui peut être négative, nous en déduisons que $Perm_P^{DL}(L)$ est illégale.

   Puisque $DDV \supset DL$ et que DL est l'abstraction la plus proche de DDV dans la hiérarchie des précisions des abstractions, DDV est minimale pour toute *permutation de boucles*.

Test de légalité associé à l'abstraction minimale :

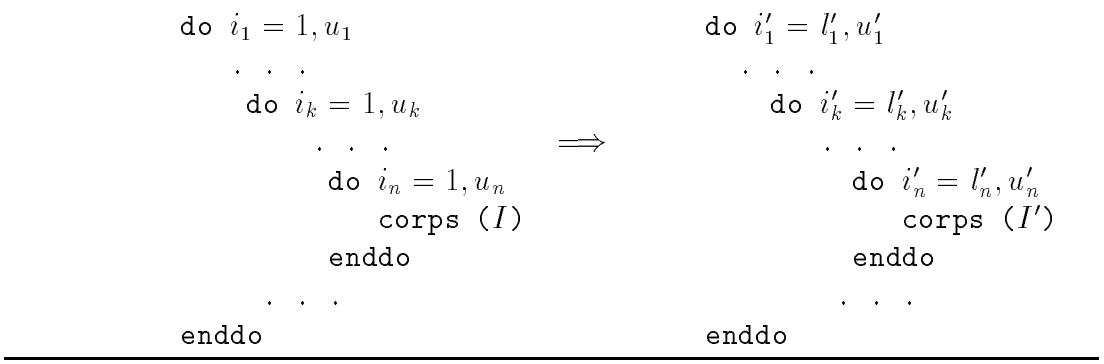
$$legal(Perm_P,\ L) \iff \forall d\vec{d}v \in DDV(L),\ Perm_P(d\vec{d}v) \gg 0$$

4.3.3 Transformation unimodulaire

Une transformation unimodulaire est une bijection d'un espace I^n dans I'^n qui peut être caractérisée par une matrice unimodulaire.

Spécification :

$TU_M(L)$ effectue une transformation unimodulaire définie par la matrice M dont le déterminant $|M|$ est égal à 1 ou -1.



où $I' = M \times I$. Les nouvelles bornes peuvent être calculées à partir du système linéaire suivant :

$$\begin{cases} A \times I \leq B \\ I = G \times I' \\ G = M^{-1} \end{cases}$$

où A, B sont deux matrices définissant les bornes de I. G, l'inverse de la matrice M, est *la matrice de la changement de base*. Un algorithme permettant de calculer les nouvelles bornes de boucles a été proposé dans [Irig88b], il a été implanté dans **PIPS** dans le cadre de cette thèse.

Effet de la transformation sur $\vec{d}$:

$$\vec{d'} = TU_M(\vec{d}) = M \times \vec{d}$$

Effet sur AD $(AD \subset D)$: Soit $P(AD) = (S, R, D)$ le polyèdre représentant les dépendances comprises dans $AD \in (DP,\ DC,\ DDV,\ DL)$. Alors

$$TU_M(P(AD)) = M \times P(AD) = (M \times S,\ M \times R,\ M \times D)$$

Test de légalité :

$$legal(TU_{\ M},\ L) \iff \forall \vec{d} \in D(L),\ M \times \vec{d} \gg 0$$

Test de légalité associé à l'abstraction AD ($AD \subset D$ **:** Soit $P(AD) = (S, R, D)$ le polyèdre représentant les dépendances comprises dans $AD \in (DP,\ DC,\ DDV,\ DL)$.

$$legal(TU_{\ M},\ L,\ AD) \iff M \times P(AD) \gg 0$$

L'abstraction admissible et minimale : pour pouvoir effectuer légalement cette transformation est : DC

Preuve :

1. DC admissible :
   Supposons que $TU^{D}_{\ M}\ (L)$ soit légale, alors
   $\forall \vec{d} \in D(L),\ \ \vec{d'} = TU_{\ M}\ (\vec{d}) = \ M \times \vec{d} \gg 0$
   $\forall \vec{dc} \in DC(L),\ \ \vec{dc} = (\lambda_1 \vec{d_1} + ... + \lambda_n \vec{d_n})\ \wedge\ (\lambda_i \geq 0,\ \sum_{i=1}^{k} \lambda_i \geq 1)$
   d'où $\ \vec{dc'} = TU_{\ M}\ (\vec{dc}) = M \times (\lambda_1 \vec{d_1} + ... + \lambda_n \vec{d_n})$
   $\lambda_i \geq 0 \Longrightarrow \vec{dc'} \gg (\sum_{i=1}^{n} \lambda_i) \times (\min_i(M \times \vec{d_i}))$[3]
   $\sum_{i=1}^{k} \lambda_i \geq 1 \Longrightarrow\ \vec{dc'} \gg \min_i(M \times \vec{d_i}) \gg 0$
   $TU^{dc}_{\ M}\ (L)$ est aussi légale.
   D'après le théorème 4.4, DC est admissible pour toute *transformation unimodulaire*.

2. DC minimale :
   Sachant que DC est admissible pour toute *transformation unimodulaire* et que
   $DC \supset DDV$ dans la hiérarchie des précisions des abstractions des dépendances présentée, nous montrons ici que l'abstraction DC est aussi minimale pour tester la légalité d'une transformation unimodulaire, en prenant un exemple pour lequel la légalité est donnée par DC et indéterminée pour une abstraction moins précise DDV.

3. où $\min_i(\textit{list de vecteurs})$ est le vecteur minimal lexicographique

Soit $L = (l_1, l_2)$ un nid de boucles et un ensemble de dépendances représentées par les deux abstractions suivantes

ii) $DC(L) = (\{(2,-1)\}, \{(2,-1),(1,0)\}, \emptyset)$

iii) $DDV(L) = \{(<,>)\}$.

Un tel cas correspond, par exemple, à une dépendance constante.

Soit $TU_M(L)$ la transformation unimodaire définie par la matrice :

$$M = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

(a) $DC(L) = (\{(2,-1)\}, \{(2,-1),(1,0)\}, \emptyset)$

$M \times DC(L) = (\{(1,-1)\}, \{(1,-1),(1,0)\}, \emptyset) \gg 0$

$\Longrightarrow TU^{DC}{}_M\ (L)$ est légale

(b) $DDV(L) = \{(<,>)\}$, $P(DDV) = (\{(1,-1)\}, \{(1,0),(0,-1)\}, \emptyset)$

$TU_M\,(DDV) = M \times P(DDV) = (\{(0,-1)\}, \{((1,0),(-1,-1)\}, \emptyset)$

$\Longrightarrow \neg(TU_M\ (DDV) \gg 0)$

$\Longrightarrow TU^{DDV}{}_M\ (L)$ est illégale.

Comme $DC \supset DDV$ et que DDV est l'abstraction dont la précision est la plus proche dans la hiérarchie des précisions des représentations des dépendances, DC est minimale pour toute *transformation unimodulaire*.

Test de légalité associé à l'abstraction minimale :

$$Soit\ DC(L) = (\cup_{1 \le i \le k} DC_i),\ legal(TU_M,\ L) \iff \wedge_{1 \le i \le k}(M \times DC_i \gg 0)$$

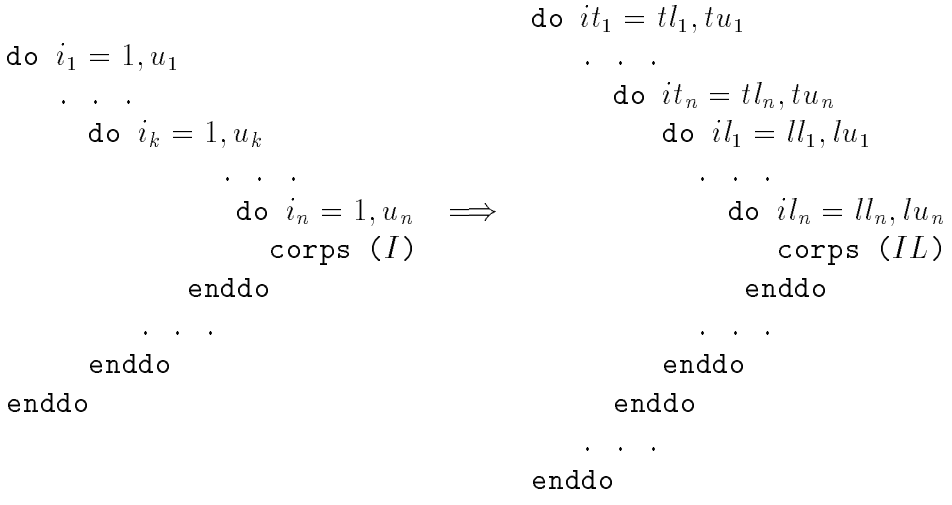
4.3.4 Partitionnement de boucles (*tiling*)

Le partitionnement d'un nid de boucles de dimension n découpe le domaine d'itérations par p familles d'hyperplans parallèles. La forme et la taille des blocs de partitionnement sont définies par `p` vecteurs rationnels $H = (\vec{h_1}, \vec{h_2}, ..., \vec{h_p})$ tels que les p plans sont respectivement orthogonaux aux `p` vecteurs de H. Cette transformation est une bijection d'un espace I^n dans I^{n+p}. Nous considerons,

dans la suite de cette section, le cas le plus géneral où $p = n$.

Spécification :

$Part_H(L)$ effectue un partitionnement caractérisé par H d'un nid de boucles L.



Deux itérations de l'espace initial i_1, i_2 appartiennent au même bloc (*tile*) si et seulement si [IrTr88a]

$$(\lfloor \vec{h_1} \times i_1 \rfloor, \lfloor \vec{h_2} \times i_1 \rfloor, ..., \lfloor \vec{h_n} \times i_1 \rfloor) = (\lfloor \vec{h_1} \times i_2 \rfloor, \lfloor \vec{h_2} \times i_2 \rfloor, ..., \lfloor \vec{h_n} \times i_2 \rfloor)$$

Test de légalité : Après partitionnement, le réordonnancement s'effectue à deux niveaux. Il faut un ordonnancement au niveau des blocs de partitionnement et un ordonnancement des itérations contenues dans chacun des blocs. Pour un bloc, les relations de dépendances entre les itérations qu'il contient sont sans cycle. Il existe donc toujours une réordonnancement des itérations à l'intérieur d'un bloc conservant toutes les relations de dépendances des itérations. L'existence d'un réordonnancement possible des blocs impose de même que les relations des dépendances entre les blocs soient partielles (où que l'exécution du bloc soit *atomique*). Deux conditions admissibles ont été proposées dans la littérature [IrTr88a].

Un partitionnement défini par H est légal ($legal(Part_H,\ L)$) si une des deux conditions suivantes est vérifiée[4] :

$$\forall \vec{d} \in D(L),\quad H \cdot \vec{d} \geq \vec{0} \qquad (1)$$

$$\forall \vec{d} \in D(L),\quad H \cdot \vec{d} \leq \vec{0} \qquad (2)$$

Test de légalité associé à l'abstraction AD ($\subset D$) : Soit $P(AD) = (S, R, D)$ le polyèdre représentant les dépendances comprises dans $AD \in (DP,\ DC,\ DDV,\ DL)$.

$$(H \cdot P(AD) \geq \vec{0}) \ \vee \ (H \cdot P(AD) \leq \vec{0}) \implies legal(Part_H,\ L,\ AD)$$

Description des conditions $H \cdot DC(L) \geq \vec{0}$ et $H \cdot DC(L) \leq \vec{0}$:

1. $H \cdot DC(L) \geq \vec{0}$ si et seulement si les conditions suivantes sont vérifiées :
   $\forall DC_i \in DC(L)$ avec $DC_i = (\{\vec{s_i}\}, \{\vec{r_j}\}, \{\vec{d_k}\})$,
   - $\forall s_i,\quad H \cdot s_i \geq \vec{0}$,
   - $\forall r_j,\quad H \cdot r_j \geq \vec{0}$,
   - $\forall d_k,\quad H \cdot d_k = \vec{0}$,
2. $H \cdot DC(L) \leq \vec{0}$ si et seulement si les conditions suivantes sont vérifiées :
   $\forall DC_i \in DC(L)$ avec $DC_i = (\{\vec{s_i}\}, \{\vec{r_j}\}, \{\vec{d_k}\})$,
   - $\forall s_i,\quad H \cdot s_i \leq \vec{0}$,
   - $\forall r_j,\quad H \cdot r_j \leq \vec{0}$,
   - $\forall d_k,\quad H \cdot d_k = \vec{0}$,

L'abstraction admissible et minimale : pour pouvoir effectuer légalement cette transformation est : DC

Preuve :

1. DC admissible :
   Supposons que $Part_H^D(L)$ soit un partitionnement légal pour lequel D

4. "$\geq \vec{0}$" implique que chaque élément est supérieur ou égale à 0

satisfait la condition (1) ci-dessus. Alors,

$\forall \vec{d} \in D(L), \quad H \cdot \vec{d} \geq \vec{0}$

Pour $\vec{dc} \in DC(L), \quad \vec{dc} = (\lambda_1 \vec{d_1} + ... + \lambda_n \vec{d_k}) \qquad (\lambda_i \geq 0, \ \sum_{i=1}^{k} \lambda_i \geq 1)$

$H \cdot \vec{dc} = (\lambda_1 H \cdot \vec{d_1} + ... + \lambda_n H \cdot \vec{d_k})$

$\lambda_i \geq 0 \Longrightarrow H \cdot \vec{dc} \geq (\sum_{j=1}^{k} \lambda_j) \cdot (\min_i(H \cdot \vec{d_j}))$

$\sum_{i=1}^{k} \lambda_i \geq 1 \Longrightarrow \ H \cdot \vec{dc} \geq \min_i(H \cdot \vec{d_j}) \geq \vec{0}$

D'après la condition (1), $Part_H^{DC}(L)$ est aussi légal.

D'après le théorème 4.4, DC est une abstraction admissible pour *le partitionnement de boucles.*

2. DC minimale :

   Sachant que DC est admissible pour *le partitionnement de boucles* et que $DC \supset DDV$, nous prenons ici un exemple pour lequel la légalité du partionnement est donnée par DC et indéterminée pour une abstraction moins précise DDV.

   Soit $L = (l_1, l_2)$ un nid de boucles et un ensemble de dépendances représentées par les deux abstractions suivantes

   i) $DC(L) = (\ \{(1,1)\}, \{(1,0),(1,1)\}, \emptyset)$

   ii) $DDV(L) = (<,<) = (\ \{(1,1)\}, \{(1,0),(0,1)\}, \emptyset)$

   et illustrées par les figures 4.1 (a), (b).

   Soit le partitionnement défini par $H = (h_1, h_2), \ h_1 = (1/3, -1/3), \ h_2 = (1/3, 0)$

   (a) $DC(L) = \{\{(1,1)\}, \{(1,0),(1,1)\}, \emptyset\}$

   $\Longrightarrow \quad H \cdot DC(L) = (\ \{(0,1/3)\}, \{(1/3,1/3),(0,1/3)\}, \emptyset)$

   Comme $s_i \ et \ r_j \in H \cdot DC(L) \geq \vec{0} \ et \ que \ d_k = \emptyset \Longrightarrow H \cdot DC(L) \geq \vec{0}$

   D'après la condition (1) du test de légalité, $Part_H^{DC}(L)$ est légal.

   (b) $DDV(L) = \{\ (<,<)\ \} = (\ \{(1,1)\}, \{(1,0),(0,1)\}, \emptyset)$

   $\Longrightarrow \quad H \cdot DDV(L) = (\ \{(0,1/3)\}, \{(1/3,1/3),(-1/3,0)\}, \emptyset)$

   $(s_1 = (0,1/3) \geq \vec{0}) \wedge r_2 = (-1/3,0) \leq \vec{0})$

   $\Longrightarrow \ \neg(H \cdot DDV(L) \leq \vec{0}) \wedge \neg(H \cdot DDV(L) \geq \vec{0})$

   $Part_H^{DDV}(L)$ est donc illégal.

Les relations de dépendances entre les blocs associés à $DC(L)$ et $DDV(L)$ sont illustrées par les figures 4.1 (c), (d).

Puisque $DC \supset DDV$, DC est minimale pour *tout partitionnement de boucles*

Test de légalité associé à DC :

$$(\wedge_{1\leq i\leq k}(H \cdot DC_i \geq \vec{0})) \ \vee \ (\wedge_{1\leq i\leq k}(H \cdot DC_i \leq \vec{0})) \implies legal(Part_H,\ L)$$

4.3.5 Parallélisation de boucles

La parallélisation est une transformation unimodulaire particulière TU_M où M est la matrice unité. Elle revient à projeter les dépendances sur un sous-espace ne comportant que les boucles *séquentielles*.

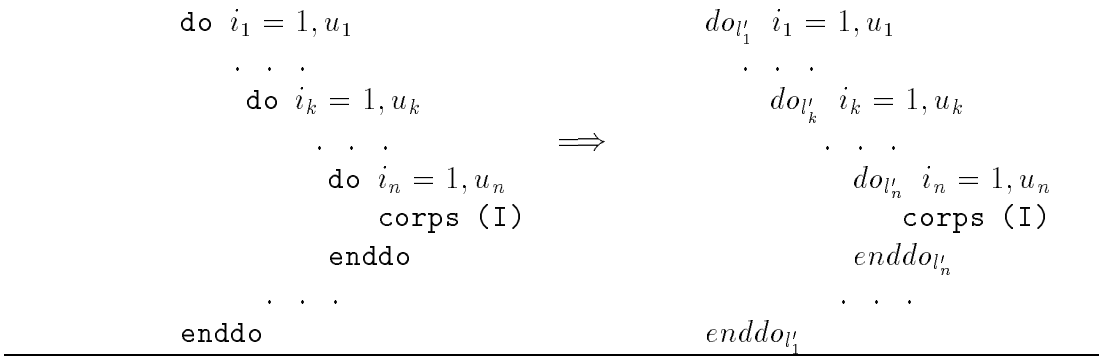
Spécification :

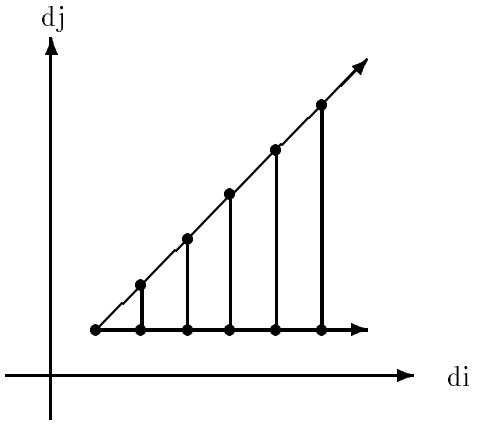
$Para_{\ PV}(L)$ parallélise le nid de boucles L selon le vecteur de parallélisation $PV = (pv[1], pv[2], ..., pv[n])$ dont les indices spécifient les boucles à paralléliser :

$$pv[k] = \begin{cases} 0 & la\ k-ieme\ boucle\ est\ parallele \\ 1 & la\ k-ieme\ boucle\ est\ sequentielle \end{cases}$$

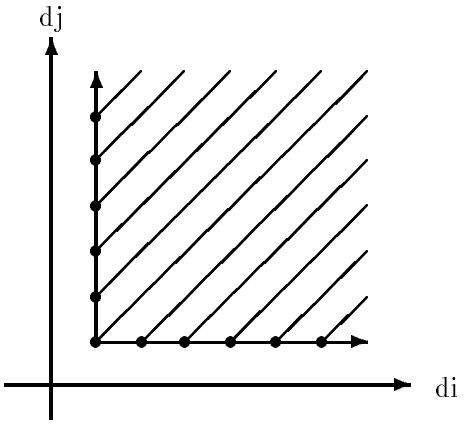
Le nid de boucles après parallélisation est illustré par la figure suivante, où

$$do_{l'_k} = \begin{cases} doall & si\ pv[k] = 0 \\ do & si\ pv[k] = 1 \end{cases}$$

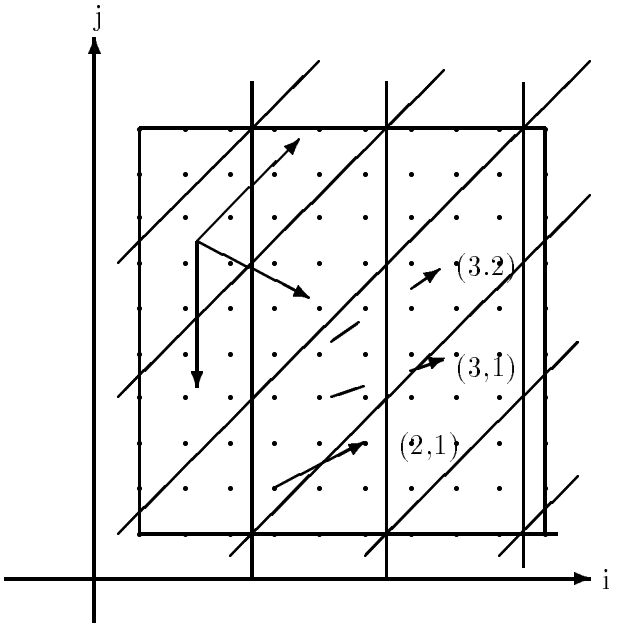




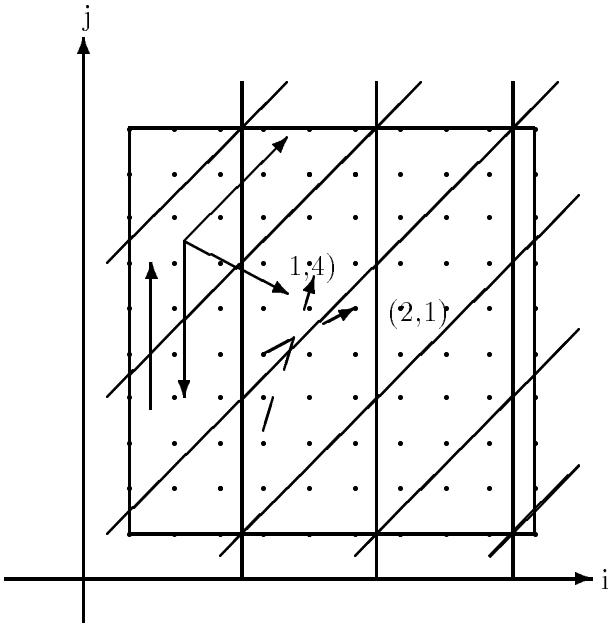
(a) DC (L)



(b) DDV (L)



(c) Le partitionnement correspondant à DC(L)



(d) Le partitionnement correspondant à DDV(L)

Note: ——→ représente la relation de dépendance entre les blocs.

- - - -→ représente la relation de dépendance entre les itérations.

FIG. 4.1 - *Le partitionnement* $H = ((1/3, -1/3/), (1/3, 0))$ *selon* $DC(L)$ *et* $DDV(L)$

Effet de la transformation sur $\vec{d}$:

$$si\ \vec{d} = (d_1, ..., d_k, ..., d_n), \quad Para\ _{PV}(\vec{d}) = PV \times \vec{d} = (d_{i_1}, ..., d_{i_k}, ..., d_{i_m})$$

où $i_1, ..., i_k, i_m$ sont les indices des boucles séquentielles.

Test de légalité:

$$legal(Part\ _{PV},\ L) \iff \forall \vec{d} \in D(L),\ Para\ _{PV}(\vec{d}) \gg 0$$

L'abstraction admissible et minimale: pour pouvoir effectuer légalement cette transformation est: DDV

Preuve:

1. DDV admissible:
   Supposons qu'il existe une transformation PV telle que $Para^{DDV}\ _{PV}(L)$ soit illégale.
   Alors, $\exists d\vec{d}v \in DDV(L)\ /\ PV \times d\vec{d}v \ll 0$.
   $\Longrightarrow\ \exists \vec{d} \in D(L)\ /\ \psi(\vec{d}) = d\vec{d}v\ et\ PV \times (\psi(\vec{d})) = PV \times (d\vec{d}v) \ll 0$ où

$$\psi(x) = \begin{cases} < & si\ x > 0 \\ > & si\ x < 0 \\ = & si\ x = 0 \end{cases}$$

   $Comme\ \ PV \times (\psi(\vec{d})) = \psi(PV \times (\vec{d}))\ \Longrightarrow\ Para\ _{PV}(\vec{d}) \ll 0$
   $\Longrightarrow\ Para^{D}\ _{PV}(L)$ est illégale.

2. DDV minimale:
   Sachant que DDV est admissible pour tester la légalité d'une *parallélisation* et que $DDV \supset DL$, nous montrons ici que l'abstraction DDV est aussi minimale pour tester la légalité de la parallélisation de boucles, en prenant un exemple pour lequel la légalité est donnée par DDV et indéterminée pour une abstraction moins précise DL.
   Soit $L = (l_1, l_2)$ un nid de boucles dont les dépendances sont représentées par les deux abstractions suivantes:
   i) $DDV(L) = \{(<, <)\}$
   ii) $DL(L) = \{\ 1\ \}$

Soit PV le vecteur de parallélisation : $(0,1)$ spécifiant que la boucle l_1 est parallèle et que la boucle l_2 est séquentielle.

(a) $DDV(L) = \{(<,<)\} \implies$
$Para\ _{PV}(DDV(L)) = PV \times DDV(L) = \{(<)\} \gg 0$
$\implies Para^{DDV}\ _{PV}(L)$ est légale.

(b) $DL(L) = \{\ 1\ \} \implies \exists \vec{d} \in D(L)\ /\ \vec{d} = (d_1, *)\ \ et\ \ d_1 > 0.$
$Para\ _{PV}(\vec{d}) = (*)$
N'ayant aucun renseignement sur la seule composante de $Para\ _{PV}(\vec{d})$, qui peut être négative, nous en déduisons que $Para^{DL}\ _{PV}(L)$ est illégale.

Puisque $DDV \supset DL$ et que DL est l'abstraction des dépendances la plus "proche" en précision de DDV, DDV est minimale pour toute *parallélisation de boucles.*

Test de légalité associé à l'abstraction minimale :

$$legal(Para\ _{PV},\ L) \iff \forall d\vec{d}v \in DDV(L),\ Para\ _{PV}(d\vec{d}v) \gg 0$$

4.4 Conclusion

Nous avons introduit, dans ce chapitre, la condition nécessaire et suffisante d'utilisation d'une abstraction AD pour tester la légalité d'une transformation TR. Nous avons montré que l'abstraction admissible et minimale

- pour toute inversion de boucle est DL,
- pour toute permutation de boucles est DDV,
- pour toute transformations unimodulaire est DC,
- pour tout partitionnement (i.e. tiling) est DC,
- pour toute parallélisation de boucles est DDV.

L'abstraction DL est généralement utilisée pour appliquer légalement la distribution de boucles car des graphes de dépendances de profondeurs différentes sont utilisés.

La transformation *Loop skewing* est toujours légale [Wolf89], mais modifie la représentation des dépendances. Pour que l'échange de boucles après une *loop skewing* permette l'obtention d'un ordonnancement optimal, la représentation des dépendances doit être précise. Quand la distance de dépendance n'est pas constante, le DDV est souvent utilisé [Wolf90], cependant DC est la représentation minimale et admissible car le *loop skewing* est une transformation unimodulaire particulière.

Certaines transformations ont besoin d'informations particulières pour pouvoir être effectuées légalement. La fusion de boucles a besoin de connaître les dépendances sur les deux boucles devant être fusionnées [Wolf91a]. La transformation *Index set splitting* a besoin de l'information : *crossing threshold* indiquant la valeur du point où l'on effectue la scission du domaine [AlKe87].

Une dépendance résultant d'une réduction doit être traitée de manière particulière car l'ordre d'exécution des opérations effectuant cette réduction n'influence pas le résultat. Le fait de savoir que la dépendance correspond à une réduction permet d'effectuer certaines transformations telles que : l'inversion de boucle et l'échange de boucles puisque le signe de la dépendance est sans importance. Il en est de même pour des opérations qui conduisent à ce type de dépendances. Les abstractions que nous avons présentées ne possèdent pas cette information. Il est possible de représenter une réduction R par un DDV étendu en élargissant l'ensemble des éléments qu'un ddv peut prendre à $\{<, =, >, \leq, \geq, *, R\}$ [Wolf91a]. Cependant, il apparait impossible de représenter l'information d'une réduction sur une (ou plusieurs) boucle(s) par un DC, car ce type d'information ne s'exprime pas par un polyèdre.

Chapitre 5

Réordonnancement mono et multidimensionnel d'un nid de boucles

Plusieurs types de parallélisation d'un programme sont possibles, entre autres, il peut être parallélisé au niveau des itérations ou des instructions. Rappelons que nous nous intéressons dans cette thèse à *la parallélisation des itérations des nids de boucles du programme.*

Dans les chapitres précédents, nous avons présenté les algorithmes de calcul des dépendances, les différentes abstractions utilisées pour les modéliser et les abstractions les plus appropriées à certains types de transformation. Pour exploiter au mieux le parallélisme implicite d'un programme une série de transformations est en général appliquée à ce programme. Une version parallèle est ensuite déduite de ce nouvel ordonnancement des boucles.

Nous avons vu dans le chapitre 4, que les transformations unimodulaires étaient tout particulièrement intéressantes pour la parallélisation des itérations. L'ensemble de ces transformations peut être caractérisé par une seule matrice unimodulaire. Le calcul d'une matrice unimodulaire permettant d'obtenir une parallélisation optimale d'un nid de boucles est plus simple que la recherche d'un ordre optimal à l'application d'une série de transformations. La méthode hyperplane est une méthode de parallélisation basée sur les transformations unimodulaires. L'exemple 1.5 illustré dans le chapitre 1 montre qu'elle est plus efficace que la méthode d'Allen & Kennedy. L'étude des méthodes de réordonnancement par

application de transformations unimodulaires est donc importante. Nous présentons, dans ce chapitre, ce type de méthodes.

La parallélisation correspond à un problème de réordonnancement des itérations d'un nid de boucles, conduisant à leur exécution parallèle. Définir l'ordonnancement d'un nid de boucles de dimension `n` consiste à trouver une *base de temps* de dimension p et une *base d'espace* de dimension q telles que $n = p+q$. Un ordonnancement de dimension p pour un nid de n boucles correspond à la génération d'un nouveau nid de boucles où les p boucles externes sont séquentielles et les $n-p$ boucles internes parallèles. Dans ce chapitre, nous présentons comment trouver un ordonnancement mono-dimensionnel et bi-dimensionnel en fonction de l'abstraction DC des dépendances contenues dans ce nid de boucles. Nous supposons, dans ce chapitre, que les boucles devant être parallélisées sont des boucles imbriquées et que l'espace de vecteurs de distance de dépendance est *full* dimensionnel [IrTr88b]. Lorsque l'espace des vecteurs de distance de dépendance n'est pas *full* dimensionnel, il existe des boucles naturellement parallèles, pour lesquelles les projections sur les vecteurs de distance sont nulles ou ont des DDV égaux à "=". Dans ce cas, on déplace ces boucles parallèles à l'extérieur, pour se ramener à un espace de vecteurs de distance de dépendance *full* dimensionnel.

5.1 Ordonnancement linéaire mono-dimensionnel

Un ordonnancement linéaire mono-dimensionnel d'un nid de n boucles correspond à un ordonnancement des boucles de manière à ce que l'ensemble des dépendances soient portées par la boucle la plus externe (qui correspond à la direction de l'ordonnancement), les `(n-1)` autres boucles étant parallèles. Ce type d'ordonnancement caractérise la *méthode hyperplane* : l'ordonnancement linéaire est défini par un vecteur $\vec{h}$, les itérations parallèles $\vec{i}$ appartiennent à des hyperplans, vérifiant $\vec{h} \cdot \vec{i} = c$, qui sont orthogonaux à la direction de l'ordonnancement.

Le schéma parallèle d'un ordonnancement mono-dimensionnel est le suivant :

```
DO i'_1 = l'_1, u'_1
   DOALL i'_2 = l'_2, u'_2
         . . .
           DOALL i'_n = l'_n, u'_n

                 corps (I')

           ENDDOALL
         . . .
   ENDDOALL
ENDDO
```

Un ordonnancement linéaire mono-dimensionnel est caractérisé par un vecteur affine $\vec{h}$ donnant la direction de l'hyperplan séquentiel. Lorsqu'on trouve un ordonnancement légal $\vec{h}$, la génération du code parallèle, précédemment présenté, revient à calculer une matrice unimodulaire $M = (m_1, ...m_n)^t$ où $m_1 = \vec{h}$ (et où la transformation du nid de boucles est représentée par $I' = M \times I$) et à calculer les bornes des nouveaux vecteurs de base. Un algorithme permettant de calculer la matrice de changement de base M à partir du vecteur d'ordonnancement $\vec{h}$ et d'obtenir les nouvelles bornes des boucles a été proposé par F. Irigoin dans [Irig88b]. Cet algorithme a été implanté dans **PIPS** dans le cadre de cette thèse.

5.1.1 Ordonnancement linéaire légal

Un ordonnancement $\vec{h}$ est **légal** s'il satisfait pour tout vecteur de distance de dépendance :

$$\forall \vec{d} \in D, \quad \vec{h} \cdot \vec{d} \geq 1 \tag{5.1}$$

L'ensemble des ordonnancements légaux H selon D est défini par :

$$H \ = \ \{\vec{h} : \ \forall \vec{d} \in D, \ \ \vec{h} \cdot \vec{d} \geq 1\} \tag{5.2}$$

Les abstractions des dépendances DC et DP étant plus faciles à calculer et plus compactes que D, nous introduisons les ordonnancements légaux selon DC et DP.

L'ensemble des ordonnancements légaux HP (respectivement HC) selon DP (respectivement DC) est défini par :

$$HP = \{\vec{hp} : \ \forall \vec{d} \in DP \ , \ \ \vec{hp} \cdot \vec{d} \geq 1\} \tag{5.3}$$

$$HC = \{\vec{hc} : \ \forall \vec{d} \in DC \ , \ \ \vec{hc} \cdot \vec{d} \geq 1\} \tag{5.4}$$

L'ensemble des ordonnancements légaux HDV selon DDV est défini par :

$$HDV = \{\vec{hv} : \ \forall \vec{d} \in D_{apr}(DDV) \ , \ \ \vec{hv} \cdot \vec{d} \geq 1\} \tag{5.5}$$

L'ordonnancement HP défini en (5.3) est équivalent à l'ordonnancement H défini en (5.2) (i.e. $H = HP$) (voir [Irig88a]). La représentation DP peut donc être aussi utilisée pour calculer un ordonnancement linéaire mono-dimensionnel légal, sans aucune perte de parallélisme. L'utilisation de DC, tel qu'il est défini dans [IrTr88b] (noté $\hat{DC}$), ne conduit pas toujours à un ordonnancement linéaire optimal (certaines boucles pouvant être séquentielles alors qu'elles peuvent être parallélisées). La nouvelle formulation de DC que nous avons proposée en 3.4 permet de résoudre ce problème. Nous illustrons cette amélioration avec l'exemple 5.1. En tenant compte de cette nouvelle définition, l'abstraction DC peut être aussi utilisée pour calculer un ordonnancement linéaire mono-dimensionnel légal, sans perte de parallélisme (i.e $HC = H$, voir le théorème 5.1). Par contre, l'utilisation de l'abstraction DDV, pour calculer un ordonnancement linéaire mono-dimensionnel légal, peut conduire à une perte de parallélisme (i.e. $HDV \subseteq H$, voir l'exemple 5.1).

Théorème 5.1 *L'ensemble des ordonancements linéaires légaux pour DC (3.4) est égal à l'ensemble des ordonnancements linéaires légaux pour D.*

Preuve

(1) Comme $D \subseteq DC$, d'après les définitions (5.2), (5.3) et 3.4 on a aussi $HC \subseteq H$.

(2) Pour prouver que $H \subseteq HC$, il suffit de prouver que chaque vecteur $\vec{h}$ de H est aussi un élément de HC.

Soit $\vec{h} \in H$. D'après la définition (5.2),

$$\forall \vec{d} \in D, \quad \vec{h} \cdot \vec{d} \geq 1$$

Soit $\vec{d}$ un vecteur quelconque de DC. D'après la définition 3.4,

$$\vec{d} = \sum_{i=1}^{k} \lambda_i \vec{d_i} \wedge \lambda_i \geq 0 \wedge \sum_{i=1}^{k} \lambda_i \geq 1$$

Alors,

$$\vec{h} \cdot \vec{d} \; = \; \sum_{i}^{k} \lambda_i \vec{h} \cdot \vec{d_i} \geq (\sum_{i}^{k} \lambda_i) \times (\min_i(\vec{h} \cdot \vec{d_i}))$$

Puisque $\sum_{i=1}^{k} \lambda_i \geq 1$,

$$\vec{h} \cdot \vec{d} \; \geq \; \min_i(\vec{h} \cdot \vec{d_i}) \; \geq 1$$

$\vec{h}$ est donc aussi un élément de HC

Résultant de (1) et (2), HC est égal à H. □

L'ensemble des ordonnancements linéaires légaux H selon D peut être repésenté par un polyèdre (voir [IrTr88b]) :

Théorème 5.2 *L'ensemble des ordonnancements linéaires légaux H selon D est un polyèdre déduit des systèmes générateurs de DP ou DC (soit ($\{\vec{s_i}\}$, $\{\vec{r_j}\}$, $\{\vec{d_k}\}$)) :*

1. *Si $\forall i \;\; \vec{s_i} \neq \vec{0}$, H est défini par :*

$$\begin{cases} \forall i, & \vec{h} \cdot \vec{s_i} \geq 1 \\ \forall j, & \vec{h} \cdot \vec{r_j} \geq 0 \\ \forall k, & \vec{h} \cdot \vec{d_k} = 0 \end{cases}$$

2. *si $\exists i \; \vec{s_i} = \vec{0}$ et s'il n'existe aucune droite ($|\{\vec{d_k}\}| = 0$) alors il n'y a pas de solution à la condition (5.1). Le cas d'un sommet nul illustre une dépendance sur une même itération, et ce type de dépendance est conservée naturellement par toutes les transformations de réordonnancement. Elle peut donc être ignorée. Le système H se définit alors par :*

$$\begin{cases} \forall i \; t.q. \; \vec{s_i} \neq \vec{0}, & \vec{h} \cdot \vec{s_i} \geq 1 \\ \forall j, & \vec{h} \cdot \vec{r_j} \geq 1 \end{cases}$$

3. Sinon $H = \emptyset$.

Lorsque les dépendances représentées par des $DDVs$ sont traduites sous forme d'un ensemble de dépendances $D_{apr}(DDV)$, defini par un système générateur, l'ensemble des ordonnancements linéaires légaux HDV peut être calculé de la même manière que HP et HC. La conversion des DDVs sous forme de système générateur a été étudiée dans le rapport [Irig88a].

Nous supposons, dans la suite, que l'exécution d'une itération prend une unité de temps et que le nombre de processeurs disponibles pour l'exécution des tâches parallèles est illimité. L'ordonnancement optimal est défini comme étant celui qui minimise le temps d'exécution total des boucles. Comme le temps d'exécution entre deux itérations i_1 et i_2 est $|h \cdot (i_1 - i_2)|$, une des conditions pour l'obtention d'un ordonnancement optimal h_{optim} est :

$$h_{optim} = \min(\max(h \cdot (i_1 - i_2))) : h \in H,\ i_1, i_2 \in I^n$$

S'il existe des paramètres au sein du domaine d'itérations I^n, l'algorithme du simplexe paramètré PIP [Feau88] est applicable.

5.1.2 Exemple

Nous illustrons ici, sur l'exemple 5.1, la recherche (1) de l'ensemble des ordonnancements linéaires légaux selon les différentes abstractions : DC, $\hat{DC}$, DDV ; (2) et le code parallèle généré avec un ordonnancement qui est optimal.

```
      DO I = 1, N
         DO J = 1, N
S1:         V(I) = W(I+J)
S2:         W(I) = V(I+J)
         ENDDO
      ENDDO
```

FIG. 5.1 - *Exemple 5.1*

1. Calcul de $DC(L)$:

   Les dépendances et leur abstraction DC pour ce nid de boucles sont :

   - $dep1 : \ V(I) \longrightarrow V(I)$

     $$Sys(di, dj) \ = \ \left\{ \ di = 0 \qquad \Longrightarrow \qquad DC(dep1) \ = \ \{\{(0,1)\}, \ \{(0,1)\}, \ \emptyset\} \right.$$

   - $dep2 : \ W(I) \longrightarrow W(I)$
     $DC(dep2) = DC(dep1)$

   - $dep3 : \ V(I+J) \longrightarrow V(I)$

     $$Sys(di, dj) \ = \ \left\{ \begin{array}{c} di \geq 1 \\ di + dj \geq 1 \end{array} \right. \qquad \Longrightarrow \qquad DC(dep3) \ = \ \{\{(1,0)\}, \ \{(0,1),(1,-1)\}, \ \emptyset\}$$

   - $dep4 : \ W(I+J) \longrightarrow W(I)$
     $DC(dep4) = DC(dep3)$

   L'union des DCs élémentaires de chacune des dépendances est définie par :

   $$DC(L) \ = \ \bigcup_{1 \leq i \leq 4} DC(depi) \ = \ \{\{(0,1)\}, \ \{(0,1),(1,-1)\}, \ \emptyset\}$$

2. Calcul de l'ensemble des ordonnancements légaux pour DC : HC est défini par le système de contraintes sur $\vec{h} = (h_1, h_2)$:

   $$\left\{ \begin{array}{l} h_2 \geq 1 \\ h_2 \geq 0 \\ h_1 - h_2 \geq 0 \end{array} \right. \qquad \Longrightarrow \qquad \left\{ \begin{array}{l} h_2 \geq 1 \\ h_1 - h_2 \geq 0 \end{array} \right.$$

   Le cône de dépendance DC et le polyèdre HC sont illustrés sur les figures 5.2 (a), (b).

3. Calcul de l'ensemble des ordonnancements pour $\hat{DC}$, $H\hat{C}$:

   Pour mettre en évidence l'amélioration apportée à la définition du cône de dépendance, proposée en 3.4, calculons maintenant $\hat{DC}(L)$ et $H\hat{C}$.

   $$\hat{DC}(L) \ = \ \{\emptyset, \ \{(0,1),(1,-1)\}, \ \emptyset\}$$

Puisque $\hat{DC}$ contient un sommet nul. Le système de contraintes pour $H\hat{C}$ est

$$\begin{cases} h_2 \geq 1 \\ h_1 - h_2 \geq 1 \end{cases}$$

$\hat{DC}$ et le polyèdre $H\hat{C}$ sont illustrés sur la figure 5.2 (c), (d).

La figure 5.2 met en évidence l'ordonnancement $h = (1, 1)$ légal pour DC mais non légal pour $\hat{DC}$.

4. Calcul de l'ensemble des ordonnancements légaux pour $DDV(L)$, HDV :

   Le DDV du nid de boucles correspond à l'union des DDVs élémentaires. Pour cet exemple :

   $$DDV(L) \ = \ \{(=, <), \ (<, *)\}$$

   $HDV(L)$ est donc l'intersection de $HDV(ddv1)$ et $HDV(ddv2)$.

   Les ordonnancements légaux pour les deux ddvs : $ddv1 = (0, <)$, $ddv2 = (<, *)$ peuvent se calculer de la façon suivante :

   - comme le système générateur de $ddv1$ est $\{\{(0, 1)\}, \{(0, 1)\}, \emptyset\}$.

     $$HDV(ddv1) = \{ \ h_2 \geq 1$$

   - comme le système générateur de $ddv2$ est $\{\{(1, 0)\}, \emptyset, \{(0, 1)\}\}$.

     $$HDV(ddv2) = \begin{cases} h_1 \geq 1 \\ h_2 = 0 \end{cases}$$

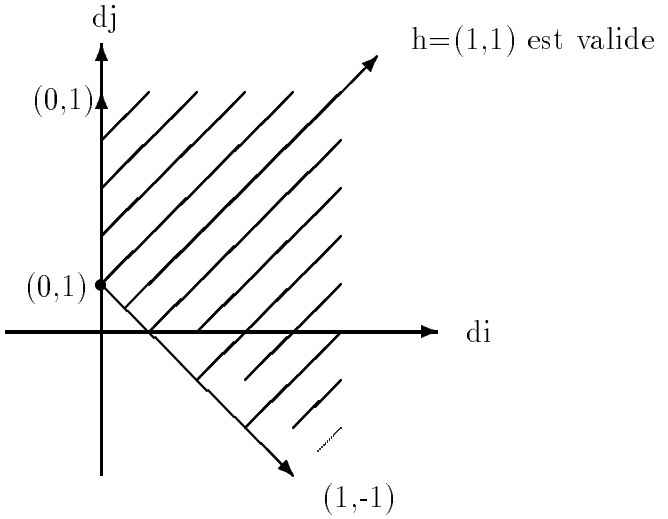
   Il n'exite donc aucun ordonnancement légal selon les DDV, car

   $$HDV(ddv1) \cap HDV(ddv2) = \emptyset$$

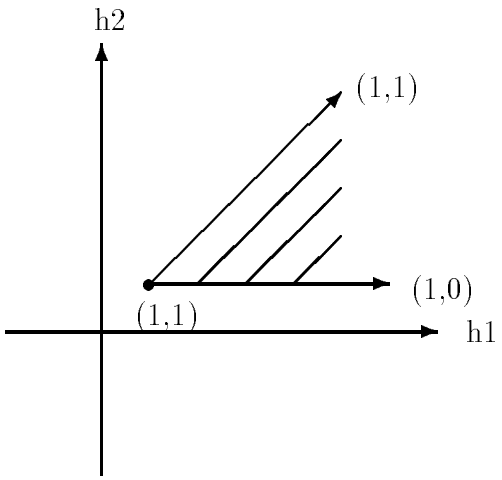
5. Génération du code parallèle pour l'ordonnancement h=(1,1).

   Dans cet exemple, l'ordonnancement $h = (1, 1)$ est légal et même optimal puisqu'il correspond au seul sommet du polyèdre H [1]. Nous donnons ici les

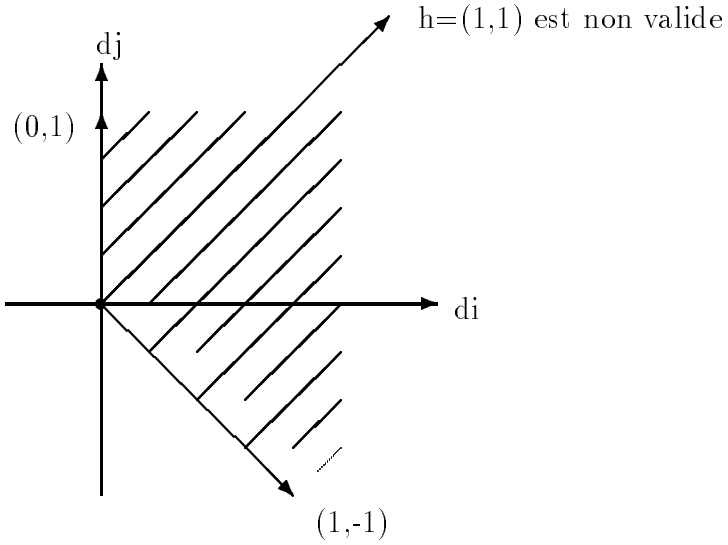
1. Si le polyèdre entier caractérisant l'ordonnancement H a un seul sommet qui est entier, il est alors l'ordonnancement optimal [Dowl90].



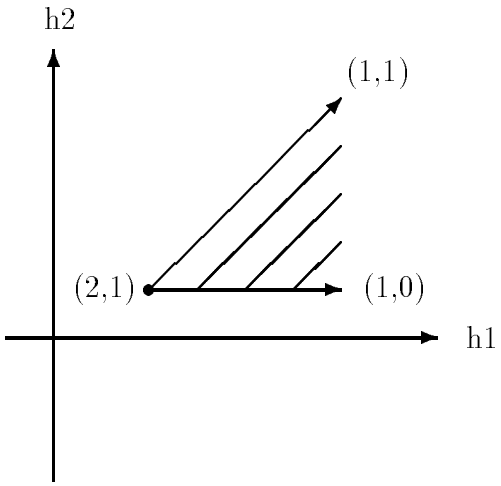
(a) DC



(b) HC



(c) $\hat{D}$



(d) $H\hat{D}$

FIG. 5.2 - *Comparaisons de DC, $\hat{DC}$ et HC, $H\hat{C}$ pour l'exemple 5.1*

résultats de **PIPS** après parallélisation du nid de boucles à partir de cet ordonnancement.

- La matrice unimodulaire de transformation est la suivante :

$$M = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix}$$

- Le code parallèle généré grâce à l'implémentation que j'ai effectuée dans PIPS est :

```
DO Ip = 2, 2*N
     DOALL Jp = MAX(1-Ip, -1*N), MIN(-1*Ip+N, -1)
         V(-1*Jp) = W(Ip)
         W(-1*Jp) = V(Ip)
     ENDDOALL
ENDDO
```

Cet exemple met en évidence que l'abstraction DDV est trop imprécise pour pouvoir être utilisée dans le cadre de la méthode hyperplane et que DC est plus précise que $\hat{DC}$. Par conséquent, le polyèdre d'ordonnancement linéaire calculé d'après DC (i.e. $H(DC)$) peut éventuellement être plus grand que celui de l'ordonnancement calculé selon $\hat{DC}$ (i.e. $H(\hat{DC})$).

5.2 Ordonnancement linéaire multi-dimensionnel

L'ordonnancement linéaire multi-dimensionnel d'un nid de boucles de dimension n correspond à un réordonnancement des boucles tel que l'ensemble des dépendances est porté par les k boucles externes ($k,\ k < n$); les $(n-k)$ autres boucles internes étant parallèles.

Pour des raisons de simplicité, nous prenons ici le cas d'un ordonnancement bi-dimensionnel. La disscusion sur l'ordonnancement multi-dimensionnel est tout a fait similaire au cas bi-dimensionnel. Le schéma parallèle de cet ordonnancement

bi-dimensionnel est le suivant :

```
DO i'_1 = l'_1, u'_1
   DO i'_2 = l'_2, u'_2
        DOALL i'_3 = l'_3, u'_3
          . . .
            DOALL i'_n = l'_n, u'_n

                  corps (I')

            ENDDOALL
          . . .
        ENDDOALL
   ENDDO
ENDDO
```

Un ordonnancement linéaire bi-dimensionnel est caractérisé par deux vecteurs affines $(\vec{h1}, \vec{h2})$ donnant la direction d'ordonnancement. L'ordonnancement $(\vec{h1}, \vec{h2})$ impose que les itérations parallèles soient dans un *hyperplan* de dimension $n-1$ vérifiant :

$$\begin{cases} \vec{h1} \cdot \vec{i} = c1 \\ \vec{h2} \cdot \vec{i} = c2 \end{cases}$$

L'ensemble des itérations parallèles forme une serie d'hyperplans parallèles de dimension $n-1$ qui sont orthogonaux à $\vec{h1}$ et à $\vec{h2}$.

Lorsqu'on trouve un ordonnancement légal $\vec{h}$, la génération du code parallèle revient à calculer une matrice unimodulaire $M = (m_1, ...m_n)^t$ où $m_1 = \vec{h1}$, $m_2 = \vec{h2}$ et à calculer les bornes des nouveaux vecteurs de base.

5.2.1 Ordonnancement légal bi-dimensionnel

Théorème 5.3 *L'ordonnancement* $(\vec{h1}, \vec{h2})$ *est* ***légal*** *si pour toutes les distances de dépendance* $\vec{d}$*, il satisfait :*

$$\forall \vec{d} \in D, \quad (\vec{h1}, \vec{h2}) \cdot \vec{d} \gg \vec{0} \tag{5.5}$$

L'ensemble des ordonnancements légaux HH d'après l'ensemble des vecteurs de distances de dépendance D se déduit de la manière suivante :

$$HH = \{(\vec{h1}, \vec{h2}) : \ \forall \vec{d} \in D \ , \ (\vec{h1}, \vec{h2}) \cdot \vec{d} \gg 0\} \tag{5.6}$$

Pour les abstractions des dépendances DP, DC et DDV, l'ensemble des ordonnancements légaux HHP, respectivement HHC, sont définis par :

$$HHP = \{(\vec{h1}, \vec{h2}) : \ \forall \vec{d} \in DP \ , \ (\vec{h1}, \vec{h2}) \cdot \vec{d} \gg 0\} \tag{5.7}$$

$$HHC = \{(\vec{h1}, \vec{h2}) : \ \forall \vec{d} \in DC \ , \ (\vec{h1}, \vec{h2}) \cdot \vec{d} \gg 0\} \tag{5.8}$$

$$HHDV = \{(\vec{h1}, \vec{h2}) : \ \forall \vec{d} \in D_{apr}(DDV) \ , \ (\vec{h1}, \vec{h2}) \cdot \vec{d} \gg 0\} \tag{5.9}$$

De même que pour l'ordonnancement mono-dimensionel, l'utilisation des abstractions DP et DC pour calculer un ordonnancement multi-dimensionnel est équivalente à celle de D.

Théorème 5.4 *Soit HHP l'ensemble des ordonnancements légaux pour DP et HHC l'ensemble des ordonnancements légaux pour DC. HHP et HHC sont équivalents à HH, l'ensemble des ordonnancements légaux pour D.*

Preuve Nous prouvons ici que HHC égal à HH.

(1) Puisque $D \subseteq DC$, d'après les définitions (5.6) et (5.7) on en déduit que $HHC \subseteq HH$.

(2) Pour prouver que $HH \subseteq HHC$, il suffit de prouver que chaque élément de HH, $(\vec{h1}, \vec{h2})$, est aussi un élément de HHC.

Soit $(\vec{h1}, \vec{h2}) \in H$. D'après la définition (5.5),

$$\forall \vec{d_i} \in D \ , \ (\vec{h1}, \vec{h2}) \cdot \vec{d_i} \gg 0$$

Soit $\vec{d}$ un vecteur quelconque de DC. D'après la définition 3.4,

$$\vec{d} = \sum_{i=1}^{k} \lambda_i \vec{d_i} \wedge \lambda_i \geq 0 \wedge \sum_{i=1}^{k} \lambda_i \geq 1$$

Alors,

$$(\vec{h1}, \vec{h2}) \cdot \vec{d} \;=\; \sum_{i}^{k} \lambda_i (\vec{h1}, \vec{h2}) \cdot \vec{d_i} \gg (\sum_{i}^{k} \lambda_i) \times (\min_i((\vec{h1}, \vec{h2}) \cdot \vec{d_i}))$$

Puisque $\sum_{i=1}^{k} \lambda_i \geq 1$,

$$(\vec{h1}, \vec{h2}) \cdot \vec{d} \;\gg\; \min_i((\vec{h1}, \vec{h2}) \cdot \vec{d_i}) \;\gg 0$$

$(\vec{h1}, \vec{h2})$ est aussi un élément de HHC

résultant de (1) et (2), HHC est égal à HH. □

L'ensemble des ordonnancements légaux bi-dimensionnels HH selon D (ou DP ou DC ou DDV) est une union de plusieurs polyèdres. Nous illustrons le calcul de HH sur l'exemple (5.2).

5.2.2 Exemple

Nous illustrons, sur l'exemple suivant, (1) le calcul d'un ordonnancement mono-dimensionnel H et d'un ordonnancement bi-dimensionnel en utilisant DP comme abstraction des dépendances; (2) la génération du code parallèle correspondante au réordonnancement bi-dimensionnel.

```
DO I = 1, N
    DO J = 1, N
        DO K = 1, N
            A(I,J,K) = A(I,J-1,K) + A(I,J,K-1) + A(I-1,N,K)
        ENDDO
    ENDDO
ENDDO
```

FIG. 5.3 - *Exemple 5.2*

1. Calcul de $DP(L)$:

   Les dépendances et leur abstraction DP pour ce nid de boucles sont les suivantes :

   - $dep1$: $A(I,J,K) \longrightarrow A(I,J-1,K)$
     $dep1$ a une distance de dépendance uniforme, (0,1,0).
     $DP(dep1) = (\{(0,1,0)\},\ \emptyset,\ \emptyset)$
   - $dep2$: $A(I,J,K) \longrightarrow A(I,J,K-1)$
     $dep2$ a aussi une distance de dépendance uniforme, (0,0,1).
     $DP(dep2) = (\{(0,0,1)\},\ \emptyset,\ \emptyset)$
   - $dep3$: $A(I,J,K) \longrightarrow A(I-1,N,K)$

   $$Sys(di,dj,dk) = \left\{ \begin{array}{l} di = 1 \\ dj \leq 0 \\ dk = 0 \end{array} \right. \Longrightarrow DP(dep3) = (\{(1,0,0)\},\{(0,-1,0)\},\emptyset)$$

   L'ensemble des dépendances représentées par DP pour le nid de boucles est éagl à l'union des DPs élémentaires pour chacune des dépendances.

   $$DP(L) = DP(dep1) \cup DP(dep2) \cup DP(dep3)$$

   Comme
   $EnvConv(\cup_{1 \leq i \leq 3} DP(dep_i)) = (\{(0,1,0),(0,0,1),(1,0,0)\},\{(0,-1,0)\},\emptyset)$
   n'est pas lexico-positive, on ne fait que l'union des polyèdres $DP(dep1)$ et $DP(dep2)$. Finalement DP est composé de deux polyèdres $DP1$ et $DP2$:

   $$DP(L) = DP1 \cup DP2$$

   $$DP1 = DP(dep1) \cup DP(dep2) = (\{(0,1,0),(0,0,1)\},\ \emptyset,\ \emptyset)$$

   $$DP2 = DP(dep3) = (\{(1,0,0)\},\ \{(0,-1,0)\},\ \emptyset)$$

2. Calcul de H pour $DP(L)$, $HP(L)$:

Soit $\vec{h} = (h_1, h_2, h_3)$ un ordonnancement linéaire. D'après le théorème 5.2, le système de contraintes devant être vérifié pour l'obtention d'un ordonnancement $\vec{h}$ légal est :

$$\begin{cases} h_1 \geq 1 \\ h_2 \geq 1 \\ h_3 \geq 1 \\ -h_2 \geq 0 \end{cases}$$

Ce système est non faisable. Il n'exite donc aucun ordonnancement linéaire mono-dimensionel pour cette boucle.

3. Calcul de HH pour $DP(L)$, $HHP(L)$:

   Soit $(\vec{h_1}, \vec{h_2})$ un ordonnancement bi-dimensionnel.

$$(\vec{h_1}\ \vec{h_2})^t = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \end{pmatrix}$$

   Nous décrivons maintenant le calcul du prédicat qui doit être vérifié sur $(\vec{h_1}, \vec{h_2})$ pour que l'ordonnancement soit légal.

   - Calcul du prédicat P_1 selon $DP1$:

     D'après la proposition 3.1

$$P_1 = (((\vec{h_1}, \vec{h_2})^t \cdot (0,1,0)) \gg 0) \wedge (((\vec{h_1}, \vec{h_2})^t \cdot (0,0,1)) \gg 0)$$

$$\begin{aligned} (\vec{h_1}, \vec{h_2})^t \cdot (0,1,0) \gg 0 &= (h_{12}, h_{22}) \gg 0 \\ &= (h_{12} \geq 1) \ \vee \ (h_{12} = 0 \wedge h_{22} \geq 1) \end{aligned}$$

$$\begin{aligned} (\vec{h_1}, \vec{h_2})^t \cdot (0,0,1) \gg 0 &= (h_{13}, h_{23}) \gg 0 \\ &= (h_{13} \geq 1) \ \vee \ (h_{13} = 0 \wedge h_{23} \geq 1) \end{aligned}$$

     Donc,

$$\begin{aligned} P_1 = \ & (h_{12} \geq 1 \wedge h_{13} \geq 1) \ \vee \ (h_{12} = 0 \wedge h_{22} \geq 1 \wedge h_{13} \geq 1) \\ & \vee \ (h_{12} \geq 1 \wedge h_{13} = 0 \wedge h_{23} \geq 1) \ \vee \ (h_{12} = 0 \wedge h_{22} \geq 1 \wedge h_{13} = 0 \wedge h_{23} \geq 1) \end{aligned}$$

– Calcul du prédicat P_2 selon $DP2$:

D'après la proposition 3.1

$$\begin{aligned} P_2 &= \{\{(h_{11}, h_{21})\}, \{(-h_{12}, -h_{22})\}, \emptyset\} \gg 0 \\ &= (h_{11} \geq 1 \wedge -h_{12} \geq 0) \ \vee \ (h_{11} = 0 \wedge h_{21} \geq 1 \wedge -h_{12} = 0 \wedge -h_{22} \geq 0) \\ &\quad \vee (h_{11} = 0 \wedge h_{21} = 0 \wedge -h_{12} \geq 1) \vee (h_{11} = 0 \wedge h_{21} = 0 \wedge -h_{12} = 0 \wedge -h_{22} \geq 1) \end{aligned}$$

– Calcul du prédicat P selon $DP(L)$:

$$\begin{aligned} P &= P_1 \wedge P_2 \\ &= (h_{11} \geq 1 \wedge h_{12} = 0 \wedge h_{13} \geq 1 \wedge h_{22} \geq 1) \\ &\quad \vee \ (h_{11} \geq 1 \wedge h_{12} = 0 \wedge h_{13} = 0 \wedge h_{22} \geq 1 \wedge h_{23} \geq 1) \end{aligned}$$

HH est donc composé de l'union de deux polyèdres. Voici les deux ordonnancements possibles correspondant aux sommets des deux polyèdres.

$$hh1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad hh2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

4. Génération du code parallèle avec l'ordonnancement $hh1$.

Une des matrices unimodulaires de transformation possible pour l'ordonnancement $hh1$ est la suivante :

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

L'application de la transformation unimodulaire M sur le nid de boucles initial conduit au code parallèle suivant :

```
DO Ip = 1, N
  DO Jp = 2, 2*N
   DOALL Kp = MAX(1,Jp-N), MIN(N, Jp-1)
    A(Ip,Jp-Kp,Kp) = A(Ip,Jp-Kp-1,Kp)+A(Ip,Jp-Kp,Kp-1)+A(Ip-1,N,Kp)
   ENDDOALL
  ENDDO
ENDDO
```

La complexité du programme initial de l'exemple 5.2 était N^3, elle est reduite à $2(N-1)^2$ après application du réordonnancement bi-dimensionnel.

5. Conséquence de l'utilisation de $EnvConv(DP(L))$:

   Sachant que

   $$EnvConv(\cup_{1\leq i\leq 3} DP(dep_i)) = (\{(0,1,0),(0,0,1),(1,0,0)\},\{(0,-1,0)\},\emptyset)$$

   n'est pas lexico-postive.

   Vérifions si les deux sommets du polyèdre caractérisant l'ensemble des ordonnancements valides dans $HH(L)$ sont légaux selon $EnvConv(DP(L))$.

   $$hh1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

   $$hh1 \cdot EnvConv(DP(L)) = (\{(0,1),(1,0),\},\ \{(0,-1)\},\ \emptyset)$$

   n'est pas lexico-positive.

   $$hh2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

   $$hh2 \cdot EnvConv(DP(L)) = (\{(0,1),(1,0),\},\ \{(0,-1)\},\ \emptyset)$$

   n'est pas lexico-positive non plus.

   Nous en déduisons que $hh1$ et $hh2$ ne sont pas des ordonnancements valides selon
   $EnvConv(DP(L))$. Ceci montre que l'utilisation de l'enveloppe convexe de D, contenant des points autres que les combinaisons convexes des éléments de D, pour calculer un ordonnancement bi-domensionnel (respectivement multi-domensionnel), conduit parfois à une perte de *scheduling* valide.

Cet exemple illustre aussi le fait que si les dépendances du nid de boucles, représentées par DP, doivent être exprimées par la composition de plusieurs polyèdres lexico-positifs (car leur enveloppe convexe est lexico-négative), alors il n'existe pas d'ordonnancement mono-dimensionnel valide. Il peut, toutefois, exister un ordonnancement bi-dimensionnel dont l'ensemble sera composé de plusieurs polyèdres. L'utilisation de leur enveloppe convexe lexico-négative peut conduire à un ordonnancement bi-dimensionnel non-valide selon $EnvConv(DP)$

5.3 Conclusion

Nous avons présenté une parallélisation globale d'un nid de boucles par application d'une méthode de réordonnancement linéaire mono- ou bi-dimensionnel à partir des abstractions DP ou DC des dépendances. Nous avons prouvé que l'utilisation de DP ou DC permettait de calculer l'ensemble des ordonnancements légaux sans perte d'information par rapport à la représentation D. L'utilisation de DP ou DC pour ce type de parallélisation, basée sur l'application d'une transformation unimodulaire, est plus efficace que celle de D, pour les cas où les dépendances ne sont pas uniformes, ou de DDV (ou DV) qui sont moins précises et encore utilisées par de nombreux prototypes [Dowl90] [WoLa91] [SaTh92].

Conclusion

Nous avons étudié, au cours de thèse, diverses phases importantes de la parallélisation d'un programme: les tests de dépendances, les abstractions des dépendances, les relations entre abstractions de dépendance et transformations de programe et le calcul d'un ordonnancement d'un nid de boucles à partir de ces abstractions.

Nous avons présenté les différentes méthodes qui ont été proposées pour trouver, de manière exacte ou approximative, les dépendances existant au sein d'un nid de boucles. Nous avons détaillé tout particulièrement l'algorithme du test de dépendance du paralléliseur **PIPS** que nous avons notablement amélioré dans le cadre de cette thèse. Une évaluation expérimentale des performances de cet algorithme a été éffectuée selon différents critères. D'après nos expériences, nous avons constaté que:

- la précision des résultats de l'analyse sémantique du programme a un impact important sur l'exactitude du test de dépendance;
- dans 93.55% des cas, les tests simples (de complexité polynômiale) suffisent pour détecter que le système de dépendances traduit une indépendance;
- un algorithme approximatif de recherche de solution entière peut être suffisant, en pratique, pour traiter les systèmes linéaires caractérisant des dépendances (notre algorithme permet l'obtention d'un résultat exact dans au moins 97.95% des cas);
- la taille des systèmes de dépendances à résoudre, lors de l'élimination d'une variabe par Fourier-Motzkin, a dans 99.73% des cas un nombre de contraintes inférieur à 4

– la complexité moyenne de l'algorithme de Fourier-Motzkin est, dans la pratique, polynômiale.

Nous avons remarqué que le manque de précision sur les dépendances, existant réellement dans le programme, vient principalement du manque d'information introduit dans les systèmes de dépendances. Ce manque d'information est dû parfois au fait que les hypothèses de linéarité ne sont pas respectées. L'utilisation de techniques non-linéaires et d'extensions interactives et dynamiques au paralléliseur sont souhaitables.

Nous avons évalué la performance des trois options du test de dépendance proposées dans **PIPS** et les améliorations qu'apportaient ces tests plus sophistiqués. Nous avons fait cette évaluation en mesurant le nombre d'indépendances trouvées pour chacune des options du test. Cependant, cette évaluation ne reflette pas l'effet du test de dépendances sur le taux de *parallélisme* engendré par chacune des options. Il faudrait compléter cette évaluation. Deux possibilités s'offrent alors: évaluer le parallélisme du programme parallélisé de manière statique en calculant le nombre de boucles parallélisées ou de manière dynamique en mesurant les temps d'exécution du programme parallèle sur une machine parallèle réelle [EiBl91] ou simulée [PePa92a] [PePa92b].

Nous avons comparé les précisions de différentes abstractions des dépendances. Leur précision est décroissante selon l'ordre suivant: les itérations de dépendance (**DI**), les vecteurs de distance de dépendance (**D**), le cône de dépendance (**DC**), les vecteurs de direction de dépendance (**DDV**) et les profondeurs de dépendance (**DL**). Nous avons ajouté une contrainte supplémentaire à la définition de l'abstraction DC caractérisant le cône de dépendance et qui a été introduite par F. Irigoin et R. Triolet dans [IrTr88b]. Cette contrainte garantit que tout *scheduling* linéaire légal déduit des informations représentées par D restera légal en utilisant cette nouvelle abstraction DC. Les algorithmes de calcul du polyèdre de dépendance et du cône de dépendance ont été intégrés au test de dépendance de **PIPS** dans le cadre de cette thèse.

Lors de l'abstraction des dépendances par un polyèdre des dépendances, il est possible de remplacer l'union de deux DP (ou DC) par leur enveloppe convexe si cette enveloppe convexe est égale à l'ensemble des éléments qui sont combinaisons

convexes des éléments des deux polyèdres. Dans les autres cas, l'utilisation de l'enveloppe convexe des dépendances ne permet pas trouver *tous* les *schedulings* linéaires valides et conduit parfois à une perte de parallélisme du programme. Des conditions suffisantes ont été donc proposées pour effectuer une union des dépendances d'un nid de boucles représentées par DP ou DC, dans tous les cas où cette union est *valide*.

Nous avons introduit la condition nécessaire et suffisante permettant d'utiliser une abstraction pour tester la légalité d'une transformation de reconstruction, de réordonnancement ou encore unimodulaire. Nous avons montré que D est une abstraction admissible pour toute transformation unimodulaire et que l'abstraction admissible et minimale 1) pour une inversion de boucle est DL, 2) pour une permutation de boucles est DDV, 3) pour une transformations unimodulaire est DC, 4) pour un partitionnement est DC, 5) pour une parallélisation de boucles est DDV. Il serait intéressant maintenant d'étudier aussi l'abstraction admissible et minimale d'autres transformations importantes telles que: *loop alignment* et *loop rotation*.

Il faut noter que la précision de DC est suffisante pour réaliser une parallélisation globale d'un nid de boucles par application d'une méthode de réordonnancement linéaire mono- ou bi-dimensionnel. L'utilisation de DP ou DC permet de calculer l'ensemble des *schedulings* légaux sans perte d'information par rapport à la représentation D.

Le calcul d'un scheduling multi-dimensionnel est un sujet récent. Plusieurs algorithmes ont été proposés par de nombreux chercheurs. Les abstractions généralement utilisées sont DI ou D. Puisque nous avons prouvé que l'utilisation de DP ou DC permet de calculer l'ensemble des *schedulings* légaux sans perte d'information par rapport à la représentation D, l'étude d'un algorithme général de calcul d'un scheduling multi-dimensionnel, utilisant DP ou DC pour les cas où D est un ensemble infini, conduirait à un résultat intéressant.

Bibliographie

[ABCC87] F. Allen, M. Burke, P. Charles, R. Cytron, J. Ferrante. "An Overview of the PTRAN Analysis System for Multiprocessing," In *Proceedings of the 1987 ACM International Conference on Supercomputing*, 1987.

[ATG91] A.-E. Al-Ayyoub, T. Terzioglu, M. Guler. "Implementation Issues of an Efficient Dependence Analysis Component for Parallelizing Compilers," In *Proceedings of the 2nd Symposium on Hight Performance Computing*, Montpellier, France, Oct. 1991.

[AlKe87] R. Allen, K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4. pp. 491-542, Oct. 1987.

[Anco91] M. Ancourt, "Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales," *Thèse de l'Université Pierre et Marie Curie*, 1991.

[Bala89] V. Balasundaram, "Interactive Parallelization of Numerical Scientific Programs," *Ph.D. Thesis, No.TR89-95*, Department of Computer Science, Rice University, Houston,Texas, July 1989.

[Bane76] U. Banerjee, "Data Dependence in Ordinary Programs," *M.S. Thesis, Rpt. No.UIUCDCS-76-837*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[Bane79] U. Banerjee, "Speedup of ordinary programs," *Ph.D. Thesis, No.UIUCDCS-79-989*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[Bane88] U. Banerjee, "Dependence Analysis for Supercomputing," Kluwer Academic Publishers, Norwell, MA, 1988.

[Bern66] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 5, Oct. 1966.

[Bran88] T. Brandes, "The importance of direct dependences for automatic parallelization," In *Proceedings of the 1988 ACM International Conference on Supercomputing,* Malo, France, July 1988.

[BuCy86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pp. 162-175, 1986.

[BePT90] H. Bennaceur, G. Plateau, F. Thomasset, "An exact algorithm for the constraint satisfaction problem: Application to dependence computing in automatic parallelization," *Rapport de INRIA*, No. 1246, Jullet 1990.

[Call87] D. Callahan, "A Global Approach to Detection of Parallelism," *Ph.D. Thesis, No. TR87-50*, Department of Computer Science, Rice University at Houston, April 1987.

[CaKe88] D. Callahan, K. Kennedy, "Analysis of Interprocedural Side Effects in a Parallel Programming Environment," *Journal of Parallel and Distributed Computing*, No. 5, pp.517-550, 1988.

[CoHa78] P. Cousot, N. Halbwachs, "Automatic Discovery of Linar Restraints Among Variables of A Program," *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, 1978.

[Cher68] N. V. Chernikova, "Algorithme for discovering the set of all the solutions of a linear programming problem," *U.S.S.R. Computational Mathematics and Mathmatical Phisics*, No. 8(6), 1968.

[Cytr84] R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," *Ph.D. Thesis*, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep. 1984.

[ChPa91] D. Y. Cheng, D. M. Pase, "An Evaluation of Automatic and Interactive Parallel Programming Tools," *Proceedings of Supercomputing'91*, Nov. 1991.

[Cray90] "CF77 Compiling System, Volume 4: Parallel Processing Guide," *SG-3074 4.0 CRAY Research*, Mendota Heights, MN, 1990.

[DaEa73] G. B. Dantzig, B. C. Eaves, "Fourier-Motzkin Elimination and its Dual," *Journal of Combinatorial Theory (A)*, No. 14, pp.288-297, 1973.

[DaRo92] A. Darte, Y. Robert, "Scheduling Uniform Loop Nests," *Rapport de Laboratoire de l'informatique du Parallélisme, Ecole Normale Superrieur de Lyon'*, No. 92-10, Fev. 1992.

[DaRR92] A. Darte, T. Risset, Y. Robert, "Loop Nest Scheduling and Transformations," *Conference on Environnement and Tools for Parallel Scientific Computing, CNRS-SNF,* Saint-Hilaire du Touvier, France, Sep. 1992.

[Dowl90] M. L. Dowling, "Optimal Code Parallelization Using Unimodular Transformations," *Parallel Computing*, No. 16, 1990.

[Duff74] R. J. Duffin, "On Fourier's Analysis of Linear Inequality Systems," *Mathematical Programming Study*, No. 1, pp.71-95, 1974, North-Holland Publishing Company.

[Duma92] A. Dumay, "Traitement des Indexations Non Linéaires en Parallélisation Automatique: Une Méthode de Linéarisation Contextuelle," *Thèse de l'Université Pierre et Marie Curie*, 1992.

[EiBl91] R. Eigenmann, W. Blume, "An Effectiveness Study of Parallelizing Compiler Techniques," In *Proceedings of 1991 International Conference on Parallel Processing*, August 1991.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Y. Li, D. Padua, "Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs," In *Proceedings of the Fourth Workshop on Programming Languages and Compilers for Parallel Computing*, California, Aug. 1991.

[Feau88] P. Feautrier, "Parametric Integer Programming," *RAIRO Recherche Operationnelle*, Sept. 1988.

[Feau89] P. Feautrier, "Asymptotically Efficient Algorithms for Parallel Architectures," *IFIP W.G. Working Conference on Decentralized Systems*, Vol.10, No. 3, Lyon, Nov. 1989.

[Feau91] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, Vol.20, No.1, 1991.

[Feau92] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part I, One-Dimensional Time," *Rapport de IBP-MASI*, No. 92.33, Avr. 1992.

[FeOW87] J. Ferrante, K.J. Ottenstein, J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987.

[Forg90] "The Forge User's Guide Version 7.01," *Pacific-Serra Research*, Dec. 1990.

[Flyn66] M. Flynn, "Very High-Speed Computing Systems," In *Proc. IEEE*, Vol.54, pp 1901-1909, 1966.

[GKT91] G. Goff, K. Kennedy, C.W.Tseng, "Practical dependence testing," In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[Gonz91] César Augusto Quiroz González, "Systematic Detection of Parallelism in Ordinary Programs," *Ph.D. Thesis* Departement of Computer Science of University of ROCHESTER, May 1991.

[GiPo88] M. Girkar, C. Polychronopoulos, "Compiling Issues for Supercomputers," In *Proceedings of Supercomputing'88*, Orlande, Nov. 1988.

[Gree71] H. Greenberg, *Integer programming.* Academic Press. NY. 1971.

[Grun90] D. Grunwald, "Data dependence analysis: The λ test revisited," In *Proceedings of 1990 International Conference on Parallel Processing* August 1990.

[Halb79] N. Halbwachs, "Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme," *Thèse de 3e cycle*, Université de Grenoble (I.N.P), 1979.

[HaPo90] M.R. Haghighat, C.D. Polychronopoulos, "Symbolic Dependence Analysis for High Performance Parallelizing Compilers," In *3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Aug. 1-3, 1990.

[HPF92] "(DRAFT) High Performance Fortran Language Specification," *High Performance Fortran Forum*, Version 0.4, Nov. 1992, Rice University, Houston Texas.

[IrTr87] F. Irigoin, R. Triolet, "Computing Dependence Direction Vectors and Dependence Cones with Linear Systems," *Rapport Interne, Ecole des Mines de Paris*, No. CAI-87-E94,1987.

[Irig88a] F. Irigoin, "Loop Reordering With Dependence Direction Vectors," In *Journées Firtech Systèmes et Télématique Architectures Futures: Programmation parallèle et intégration VLSI*, Paris, Nov. 1988.

[Irig88b] F. Irigoin, "Code Generation for the Hyperplane Method and for Loop Interchange," *Rapport Interne, Ecole des Mines de Paris*, No. CAI-88-E102, 1988.

[IrTr88a] F. Irigoin, R. Triolet, "Supernode Partitioning," In *Conference Record of Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[IrTr88b] F. Irigoin, R. Triolet, "Dependence Approximation and Global Parallel Code Generation for Nested Loops," In *International Workshop Parallel and Distributed Algorithms*, Bonas, France, Oct. 1988.

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

[Irig92] F. Irigoin, "Interprocedural Analyses For Programming Environments," In *Workshop on Environments and Tools For Paralle Scientific Computing, CNRS-SNF,* Saint-Hilaire du Touvier, France, Sep. 1992.

[Kuck77] D. Kuck, "A Survey of Parallel Organization and Programming," *ACM on Computing Surveys*, March 1977.

[Kuck78] D. Kuck, *The Structure of Comuters and Computations*, Vol. 1, John Wiley and Sons, New York, 1978.

[Kuck84] D. Kuck et. al., "The Effects of Program Restructuring, Algorithme Change and Architecture Choice on Program Performance," In International Conference on Parallel Processing, Aug. 1984.

[Kuck89] Kuck & Associates, "KAP/CRAY User's Guide," University of Illinois at Urbana-Champaign,, IL, 1989.

[Kuhn80] R.H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees," *Ph.D. Thesis, Rpt. No. UIUCDCS-R-80-1009,* Department of Computer Science, University of Illinois at Urbana-Champaign, Feb. 1980.

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

[KKP90] X.Y. Kong, D. Klappholz, K. Psarris, "The I Test: A New Test for Subscript Data Dependence," In *Proceedings of 1990 International Conference on Parallel Processing*, Padua, August 1990.

[KPK90] D. Klappholz, K. Psarris, X.Y. Kong, "On The Perfect Accuracy of an Approximate Subscript Analysis Test," In *Proceedings of the 1990 ACM International Conference on Supercomputing,* June 1990.

[KMT91] K. Kennedy, K.S. McKinley, C.-W. Tseng, "Analysis and Transformation in the ParaScope Editor," In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KrSa91] V.P. Krothapali, P. Sadayappan, "Removal of Redundant Dependences in DOACROSS Loops With Constant Dependences," In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, April 21-24, 1991.

[Lamp74] L. Lamport, "The Parallel Execution of DO Loops," *Communications odf the ACM*, Vol. 17, No. 2, Feb. 1974.

[LiAb87] Z.Y. Li, W. Abu-Sufah, "On Reducing Data Synchronization in Multiprocessed Loops," *IEEE Transactions on Computers*, Vol. C-36, No. 1, Jan. 1987.

[LuCh91] L.-C. Lu, M.C. Chen, "Parallelizing Loops with Indirect Array References or Pointers," In *Proceedings of the Fourth Workshop on Programming Languages and Compilers for Parallel Computing*, California, Aug. 1991.

[LCD91] D. Levine, D. Callahan, J. Dongarra, "A Comparative Study of Automatic Vectorizing Compilers," *Parallel Computing*, No, 17, 1991.

[LiTh88] A. Lichnewsky, F. Thomasset, "Introducing Symbolic Problem Soving Techniques in the Dependence Testing phases of a Vectorizer," In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pp. 396-406, July 1988.

[LiYe89] Z.Y. Li, P.-C. Yew, "Some results on Exact Data Dependence Analysis," In *2nd Workshop on Programming Languages and Compilers for Parallel Computing,* 1989.

[LYZ89] Z.Y. Li, P.-C. Yew, C.Q. Zhu , "Data dependence analysis on multi-dimensional array references," In *Proceedings of the 1989 ACM International Conference on Supercomputing*, pp. 215-224, Crete, Greece, June 1989.

[Li91] Z.Y. Li, "Compiler Algorithms for Event Variable Synchronization," In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[MHL91] D. Maydan, J.L. Hennessy, M.S. Lam, "Efficient and Exact Data Dependence Analysis," In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[Mayd92] D.E. Maydan, "Accurate Analysis of Array References," *Ph.D. Thesis, The Department of Computer Science of Stanford University*, Sep. 1992.

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

[MiPa87] S. P. Midkiff, D. Padua, "Compiler Algorithms for Synchronization," *IEEE Transactions on Computers*, Vol. C-36. No. 12. Dec. 1987.

[Padu79] David Alejandro Padua Haiek, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," *Ph.D. Thesis, Rpt. No. UIUCDCS-R-79-990*, Department of Computer Science, University of Illinois at Urbana-Champaign, Nov. 1979.

[PeCy87] J.-K. Peir, R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," In *Proceedings of 1990 International Conference on Parallel Processing*, August 1987.

[PePa91] Paul Petersen, David Padua, "Experimental Evaluation of Data Dependence Tests (Extended Abstracts)," *CSRD Report No. 1080*, Feb. 1991.

[PePa92a] P. Petersen, D. Padua, "Machine-Independent Evaluation of Parallelizing Compilers," *CSRD Report No. 1173*, Jan.1992.

[PePa92b] P. Petersen, D. Padua, "Dynamic Dependance Analysis: A Novel Method for Data Dependence Evaluation," *5th Annual Workshop on Languaegs and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[Plat90] A. Platonoff, "Calcul des Effets des Procédures au Moyen des Régions," *Rapport Interne, Ecole des Mines de Paris*, No. EMP-CAI-I 132, 1990.

[Poly88] C. D. Polychronopoulos, *Parallel Programming and Compilers,* Kluwer Academic Publishers, U.S.A, 1988.

[PoGi89] C.D. Polychronopoulos, M. Girkar et al., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," In *Proceedings of 1989 International Conference on Parallel processing*, pp. II.39-II.48, 1989.

[PKK91] K. Psarris, X.Y. Kong, D. Klappholz, "Extending The I Test to Direction Vectors," In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Pugh92] W. Pugh, "The Omega Test: a fast and practical integer programming algorithm for dependence analysis," In *Communications of the ACM*, Aug. 1992.

[RaSa90] J. Ramanujam, P. Sadayappan, "Tiling of Iteration Spaces for Multicomputers," In *Proceedings of 1990 International Conference on Parallel processing*, pp. II.179-II-186, 1990

[Riba90] H.B. Ribas, "Obtaining Dependence Vectors for Nested-Loop Computations," In *Proceedings of 1990 International Conference on Parallel Processing*, August 1990.

[Rose90] C. M. Rosene, "Incremental Dependence Analysis," *Ph.D. Thesis, No.TR90-112*, Department of Computer Science,Rice University, Houston,Texas, March, 1990.

[Sark90] V. Sarkar, "PTRAN — the IBM Parallel Translation System," In *Proceedings of International Workshop on Compilers for Parallel Computers*, Paris, Dec. 1990.

[SMC91] J.H. Saltz, R. Mirchandaney, K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Transactions on Computers*, Vol. 40, No. 5, May 1991.

[SaTh92] V. Sarkar, R. Thekkath, "A General Framework for Iteration-Reording Loop Transformations," In *SIGPLAN'92 Conference on Programming Language Design and Implementation,* San Francisco, JUNE, 1992.

[Schr86] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons 1986.

[SLY89] Z.Y. Shen, Z.Y. Li, P.-C. Yew, "An Empirical Study on Array Subscripts and Data Dependencies," In *Proceedings of 1989 International Conference on Parallel processing*, pp. 145-152, 1989.

[Shos81] R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," *Journal of the Association for Computing Machinery*, Vol.28., No. 4, Oct. 1981.

[ShFo91] W. Shang, J. A. B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, Vol. 40, No. 6, June 1991.

[Tarj72] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.*, Vol.1, No.2, pp.146-160, June 1972.

[TaFa92] N. Tawbi, P. Feautier, "Processor Allocation and Loop Scheduling on Multiprocessor Computers," In *Proceedings of the 1987 ACM International Conference on Supercomputing*, 1992.

[Trio84] R. Triolet, "Contribution à la Parallélisation Automatique de Programmes Fortran Comportant des Appels de Procédure," *Ph.D. Thesis*, Université Pierre et Marie Curie, 1984.

[TIF86] R. Triolet, F. Irigoin, P. Feautrier, "Direct Parallelization of CALL statements," In *Proceedings of the ACM SIGPLAN'86 Symp. on Compiler Construction*, Vol. 21, No.6, June 1986.

[TDF87] N. Tawbi, A. Dumay, P. Feautrier, "PAF: un Paralléliseur Automatique pour FORTRAN," *Rapport Interne de MASI*, No. 185, Université Pierre et Marie Curie, 1987.

[TWLPH91] S. Tjiang, M. Wolf, M. Lam, K. Pieper, J. Hennessy. "Integrating Scalar Optimization and Parallelization," In *Proceedings of the Fourth Workshop on Programming Languages and Compilers for Parallel Computing*, California, Aug. 1991.

[Wall88] D.R. Wallace, "Dependence of Multi-Dimensional Array References," In *Proceedings of the 1988 ACM International Conference on Supercomputing*, Malo,France, July 1988

[WaEi93] J. Wang, C. Eisenbels, "Decomposed Software Pipelining: A New Approach to Exploit Instruction Level Parallelism for Loop Programs," In *Proceedings of Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, Jan. 1993.

[Will76] H. P. Williams. "Fourier-Motzkin Elimination to Integer Programming Problems," *Journal of combinational theory (A)*, 21, pp. 118-123, 1976.

[WoBa87] M. Wolfe, U. Banerjee. "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming*, Vol. 16, No. 2, 1987.

[WoLa90] M.E. Wolf, M.S. Lam, "Maximizing Parallelism via Loop Transformations," In *3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Aug. 1-3, 1990.

[WoLa91] M.E. Wolf, M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.4, Oct. 1991.

[Wolf89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, Londin, 1989.

[Wolf89b] M. Wolfe, " Loop Rotation," In *2nd Workshop on Programming Languages and Compilers for Parallel Computing,* 1989.

[Wolf90] M. Wolfe, "Experiences with Data Dependence and Loop Restructuring in the Tiny Research Tool," *Technical Report*, No. CS/E 90-016, Sep. 1990.

[Wolf91a] M. Wolfe, "Experiences with Data Dependence Abstractions," In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Wolf91b] M. Wolfe, "The Tiny Loop Restructuring Research Tool," In *Proceedings of 1991 International Conference on Parallel Processing*, August 1991.

[WoTs91] M. Wolfe, C.W. Tseng, "The Power Test for Data Dependence," *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[Yang92] Y.Q. Yang, "Evaluation Expérimentale de Tests de Dépendance," *4eme Rencontre du Parallelisme* Lille, Mar. 1992.

[ZiCh90] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*, Acm Press, Addision-Weslxy Publishing, New York, 1990.